COORDINATED HIGHWAYS ACTION RESPONSE TEAM
STATE HIGHWAY ADMINISTRATION

FMS R1B2 Detailed Design

Contract DBM-9713-NMS
TSR # 9803444
Document # M303-DS-003R0

July 18, 2000
By
Computer Sciences Corporation and PB Farradyne Inc

Revision	Description	Pages Affected	Date
0	Initial Release	All	July 18, 2000

Table of Contents

List of Figures

1 Introduction

1.1 Purpose

This document describes the detailed design of the FMS subsystem for Release 1, Build 2. This design provides the details for the high level design presented in document M303-DS-003R0, ***"FMS R1B2 High Level Design."***

1.2 Objectives

The main objective of this design is to provide software developers with details regarding the implementation of the software included in the FMS subsystem for R1B2. This document also serves to provide documentation to those outside the software development community to show how the field communications requirements of CHART II are being accounted for in the software design.

1.3 Scope

This design is limited to Release 1, Build 2 of the FMS subsystem and to the field communications specific requirements of the CHART II System Requirements Specification. Although the FMS R1B2 High Level Design includes the design for incorporating HAR and SHAZAM device communications into the FMS subsystem, this detailed design does not address communications for these devices. The detailed design for these devices is planned for the R1B3 release.

1.4 Design Process

As in the high level design, object-oriented analysis and design techniques were used in creating this design. As such, much of the design is documented using diagrams that conform to the Unified Modeling Language (UML), an industry standard for diagramming object-oriented designs.

In the high level design, system interfaces were identified and specified. These interfaces were partitioned into logical groupings of packages. This design serves to fill in the details necessary to implement each of the system interfaces identified in the high level design.

In this design, each package identified in the high level design is addressed separately with its own class diagram and sequence diagrams for major operations included in the package's interfaces. Additionally, packages needed for implementation but not present in the high level design are included in this design, with each of these also having its own class diagram and sequence diagrams. Packages are also included for third party software that is needed by the CHART II software, such as the ORB and Java classes. Only classes and methods shown on the sequence diagrams are included in diagrams for third party products.

The design process for each package involved starting with a class diagram including interfaces from the high level design, and filling in details to the class diagram to move toward implementation. Sequence diagrams were then used to show how the functionality is to be carried out. An iterative process was used to enhance the class diagram as sequence diagrams identified missing classes or methods.

1.5 Design Tools

The work products contained within this design are extracted from the COOL:JEX design tool. Within this tool, the design is contained in the CHART II project, R1B2 configuration, System Design phase. The following system versions contain designs relating to the FMS Subsystem:

- **System Interfaces**
  This package contains a class diagram named FieldCommunications that specifies the IDL for the FMS subsystem. This IDL will be used to generate the code for the Field Communications package that contains the code that allows objects to be accessed remotely via an ORB.
- **Field Communications Module**
  This package contains the design for the implementation of the objects and supporting classes to serve the interfaces defined in the Field Communications package.
- **DMS Protocols**
  This package contains the design of Protocol Handlers for DMS devices. These classes utilize the DataPort interface specified in the FieldCommunications package to perform communication sequences for a specific device type and model.
- **Device Utility**
  This package was created as part of the R1B2 CHART II Servers Detailed Design and contains utility classes that are used by device objects. The PortLocator class has been added to this package because device objects will also use it.
- **Java Classes**
  This package contains classes that are part of the Java programming language referred to by this design.
- **CORBA Utilities**
  This package contains CORBA classes that are referred to by this design.

1.6 Work Products

This design contains the following work products:

- A UML Class diagram for each package showing the low level software objects that will allow the system to implement the interfaces identified in the high level design.
- UML Sequence diagrams for non-trivial operations of each interface identified in the high level design. Additionally, sequence diagrams are included for non-trivial methods in classes created to implement the interfaces. Operations that are considered trivial are operations that do nothing more than return a value or a list of values and where interaction between several classes is not involved.

2 Key Design Concepts

2.1 Service Application Module

The FMS subsystem utilizes the CHART II service application framework to serve its CORBA objects. The FieldCommunicationsModule is a CHART II service application module that can be included in a Chart2Service application. The Chart2Service provides the main entry point for a service application and through its use of a ServiceApplication utility provides many services that are useful for applications that serve CORBA objects.

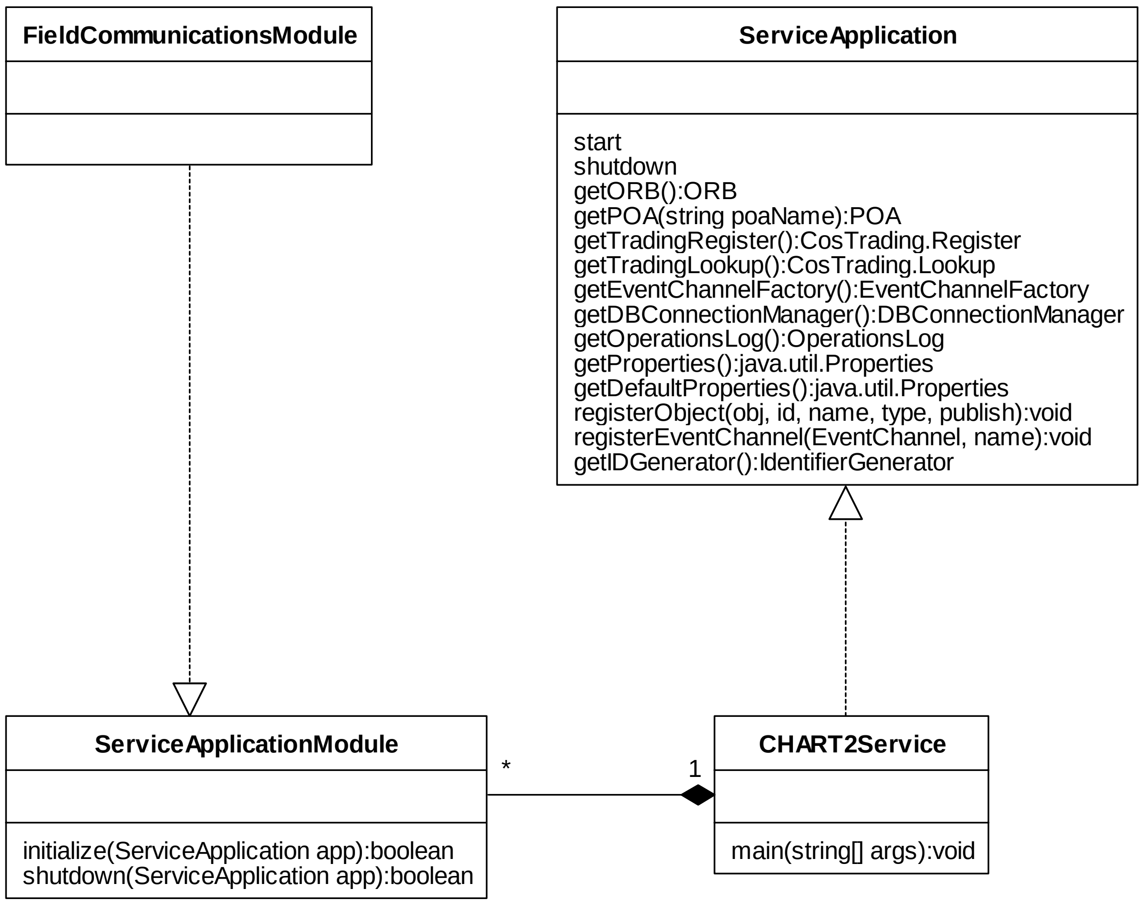

FieldCommunicationsModule
ServiceApplication
start
shutdown
getORB():ORB
getPOA(string poaName):POA
getTradingRegister():CosTrading.Register
getTradingLookup():CosTrading.Lookup
getEventChannelFactory():EventChannelFactory
getDBConnectionManager():DBConnectionManager
getOperationsLog():OperationsLog
getProperties():java.util.Properties
getDefaultProperties():java.util.Properties
registerObject(obj, id, name, type, publish):void
registerEventChannel(EventChannel, name):void
getIDGenerator():IdentifierGenerator
ServiceApplicationModule
initialize(ServiceApplication app):boolean
shutdown(ServiceApplication app):boolean
*
1
CHART2Service
main(string[] args):void

Figure 1. Service Application Module (Class Diagram)

The FieldCommunicationsModule takes advantage of the services provided by the service application framework, such as publication and clean up of objects in the CORBA trader, access to the ORB's POA, access to a DB connection manager object and an IdentifierGenerator. Furthermore, because the FieldCommunicationsModule is a standard CHART II service application module, it can be deployed in the same Java virtual machine as other service application modules, providing efficiency in calls between objects in these modules. This would be desirable for an object that has heavy communications usage, such as an automated data recorder (ADR).

2.2 Module Initialization

The FieldCommunicationsModule, like all CHART II service application modules, is initialized when the Chart2Service application is started. The FieldCommunicationsModule creates a single PortManager object, activates the object to make it available for CORBA requests, and publishes the object in the CORBA trader (using the service application's registerObject() method).

When the PortManager object is created, it creates Port objects for each communication port for which it is configured to provide access. To keep the PortManager code generic and not dependent on any specific type of Port object, the PortManager uses the InstallablePort interface to instantiate and initialize port objects.

Part of the configuration information for a Port is the name of the class that provides the implementation for the specific type of communications port. The PortManager instantiates each port object using the specified class name. Because each Port object, regardless of its derived type, must support the InstallablePort interface, the PortManager can call the init() method on any port object, regardless of the derived type. When a Port's init() method is called it performs initialization that is specific to the derived type of the object.

2.3 Port Management

The PortManager object manages access to Port objects through lists that group ports according to their type. When the demand for ports from a PortManager is greater than its supply, the PortManager uses a wait list to provide prioritized access to the next available port.

2.3.1 Port Lists

The PortManager uses three separate lists to store the Port objects that it manages. The free list is used to store ports that are currently free to be given to a client when requested. The in-use list is used to store ports that have been given to a client for use but not yet released. A third list, the marginal list, is used to store ports that are free, but have experienced an error during their last use.

When a port is requested for use, the PortManager first looks for a free port in its free list. If a port is not available, the PortManager looks for a port in its marginal list. If there are no ports in the marginal list, the PortManager adds an entry to the wait list and waits the specified amount of time for the wait list to notify that a port has become available.

2.3.2 Wait List

The wait list is used to return ports to requesters based on their priority and the time of their request. All requests of the same priority are served in a first come, first served order. When a port is not immediately available upon request, the PortManager adds an entry to the wait list and executes a wait on that object. When a port is released, the PortManager notifies only the highest priority entry in the wait list and removes the entry from the wait list.

2.4 Port Reclaiming

The PortManager periodically checks each port in the in-use list to determine if the port has exceeded its inactivity threshold. If the port is deemed to be inactive, the PortManager deactivates the object in the ORB that disables the port for future use by the client that holds a reference to the port. The port manager then removes the port from the in-use list and adds the port to free or marginal list, as appropriate. Lastly the PortManager notifies the wait list that a port is available.

The PortManager uses a timer object to periodically execute the PortReclaimer timer task, which delegates its processing to the PortManager, thus allowing all port management to be done by the PortManager. The period used for the timer is configurable and exists in the FieldCommunicationsProperties object.

2.5 Port Implementations

The R1B2 release of CHART II requires only modem communications to field devices (ISDN and POTS), however the FMS subsystem is also implementing the DirectPort interface because its code is reusable by the implementation of the ModemPort, which inherits all of the direct port’s functionality. Existence of this direct port implementation provides greater options for testing the FMS subsystem with a limited number of ISDN and POTS lines.

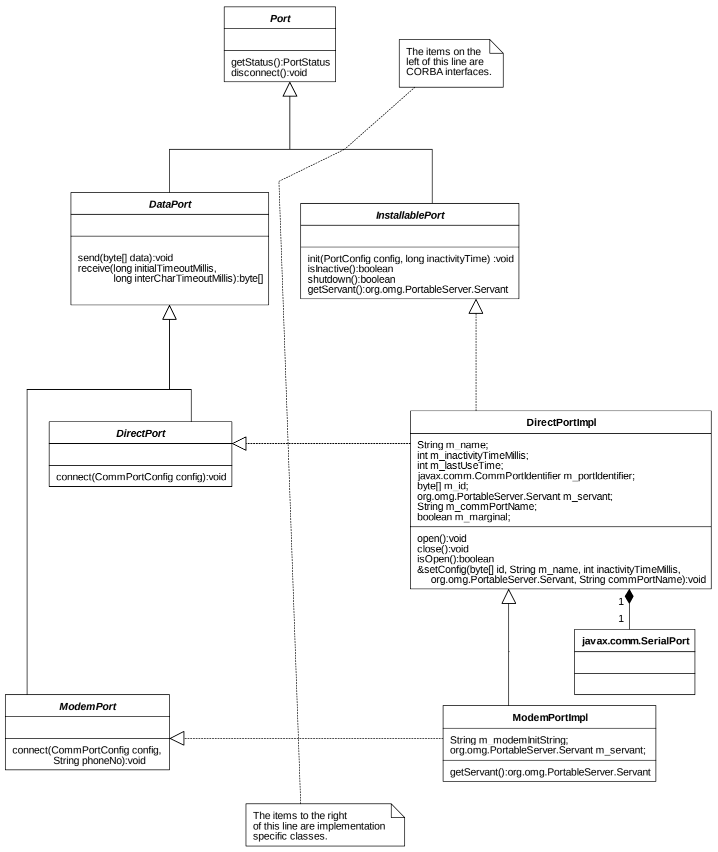

Port
getStatus():PortStatus
disconnect():void
The items on the left of this line are CORBA interfaces.
DataPort
send(byte[] data):void
receive(long initialTimeoutMillis, long interCharTimeoutMillis):byte[]
InstallablePort
init(PortConfig config, long inactivityTime) :void
isInactive():boolean
shutdown():boolean
getServant():org.omg.PortableServer.Servant
DirectPort
connect(CommPortConfig config):void
DirectPortImpl
String m_name;
int m_inactivityTimeMillis;
int m_lastUseTime;
javax.comm.CommPortIdentifier m_portIdentifier;
byte[] m_id;
org.omg.PortableServer.Servant m_servant;
String m_commPortName;
boolean m_marginal;
open():void
close():void
isOpen():boolean
&setConfig(byte[] id, String m_name, int inactivityTimeMillis, org.omg.PortableServer.Servant, String commPortName):void
1
1
javax.comm.SerialPort
ModemPort
connect(CommPortConfig config, String phoneNo):void
ModemPortImpl
String m_modemInitString;
org.omg.PortableServer.Servant m_servant;
getServant():org.omg.PortableServer.Servant
The items to the right of this line are implementation specific classes.

Figure 2. Port Implementation (Class Diagram)

The DirectPortImpl class implements the DirectPort connect method to open a serial port on the PC (e.g. COM1) and also provides methods to send and receive bytes on the serial port. The ModemPortImpl simply provides a connect method that first calls its base class connect method to open the serial port and then uses the base class send and receive methods to interact with the modem to establish a connection. After a connection is established, the base class send and receive methods allow the user of the port to exchange data with the remote device.

2.6 Port Locator

The PortLocator is a utility class that implements fail over for clients of PortManagers. The PortLocator is provided an ordered list of PortManagers during its creation. The PortLocator provides a getPort method with the same signature as an actual PortManager. The getPort method is responsible for finding object references in the CORBA Trader and attempting to get a port from the first port manager on the list. When the port is returned from the PortManager, the PortLocator returns the port to its caller and stores a reference to the PortManager from which the port was retrieved. When the PortLocator is requested to release the port, the PortLocator calls the releasePort method of the PortManager where the port was earlier retrieved.

In the event that a call from the PortLocator to the PortManager's getPort method should fail, the PortLocator executes fail over processing based on the fail over options set in the PortLocator. The following error types may occur:

1. The PortManager cannot be called to get a port. The PortManager may be down or a network problem may exist that prevents the call from being delivered.

2. The PortManager gets the request for a port but the port manager does not have any ports of the specified type under its management.

3. The PortManager has a greater demand for the specified port type than it has supply and a port does not become available within the timeout period specified in the getPort call.

4. An unexpected error is encountered within the PortManager processing.

Settings exist in the PortLocator for each of the above error conditions to determine whether or not the PortLocator should fail over. When the PortLocator has encountered an error for which it is set to fail over, the PortLocator repeats its attempt to retrieve a port using the next PortManager in its list.

Additional settings exist in the PortLocator to allow retries to be done on the first PortManager prior to failing over to another PortManager. The number of retries is specified as well as the failure conditions for which retries should be done. Note that retries are only done for the first PortManager in the list. After fail over to secondary PortManagers has begun, retries are not used.

2.7 Protocol Handlers

Protocol handler implementations vary based on the protocol that they support. Some general concepts apply to all DMS protocol handlers. All protocol handlers perform their communications using a DataPort object, which is a port (direct connect or modem) that allows binary data to be sent and received. All protocol handlers must handle the task of converting DMS messages to and from the MULTI format.

When a message is to be set on a DMS, it is passed to the protocol handler in MULTI format. The protocol handler must interpret the MULTI to determine which characters to put at each location on the sign display and handle multiple page messages, page timing, etc. While much of this task is specific to the protocol being implemented, a common utility class named MultiConverter is used to parse a MULTI string and notify a listener of the high level constructs in the message, such as new line, new page, and justification. Each DMS protocol handler implements the MultiParseListener interface so it can be directly notified of the format specified by a MULTI Message.

When the status is retrieved from a DMS, the protocol handler must return the sign's current message in MULTI format. Because the DMSs supported by R1B2 do not support MULTI directly, protocol handlers must convert the message retrieved from the sign from the typical byte array(s) into MULTI. To prevent each protocol handler from having to code this functionality, an intermediate format has been defined in the utility class DMSHardwarePage. This class represents one page of the physical DMS display and contains a two dimensional array whose size matches the physical row/column size of the DMS. Protocol handlers fill these page objects with the exact text retrieved from the DMS (including blanks). After the current message is put in the DMSHardwarePage format, it can be passed to the MultiConverter to convert the text as laid out on the DMS into MULTI, including justification tags.

2.8 Database

The database requirements for FMS R1B2 are minimal. The database is only used for start-up configuration data. A table exists for generic port information to allow the PortManager to generically instantiate the port objects. A table exists for each specific port type (DirectPort and ModemPort) to provide configuration values that are specific to the port type. The following schema is used:

Table Port

Key	Column Name	Column Type
*	Port_ID	CHAR(32)
	Port_Name	VARCHAR(128)
	Port_Type	NUMBER(5)
	Class_Name	VARCHAR(128)

Table DirectPort

Key	Column Name	Column Type
*	Port_ID	CHAR(32)
	Com_Port_Name	VARCHAR(64)

Table ModemPort

Key	Column Name	Column Type
*	Port_ID	CHAR(32)
	Com_Port_Name	VARCHAR(64)
	Init_String	VARCHAR(64)

At startup, the PortManager reads all rows from the Port table and instantiates an object for each row using the class name specified. Each port object is initialized, at which time it reads its type specific data from the appropriate table (DirectPort or ModemPort) and performs other initialization processing.

3 Package Designs

3.1 Field Communications (IDL)

This diagram shows system interfaces relating to field communications. These interfaces, typedefs, and enums specify the IDL for the FieldCommunications package.

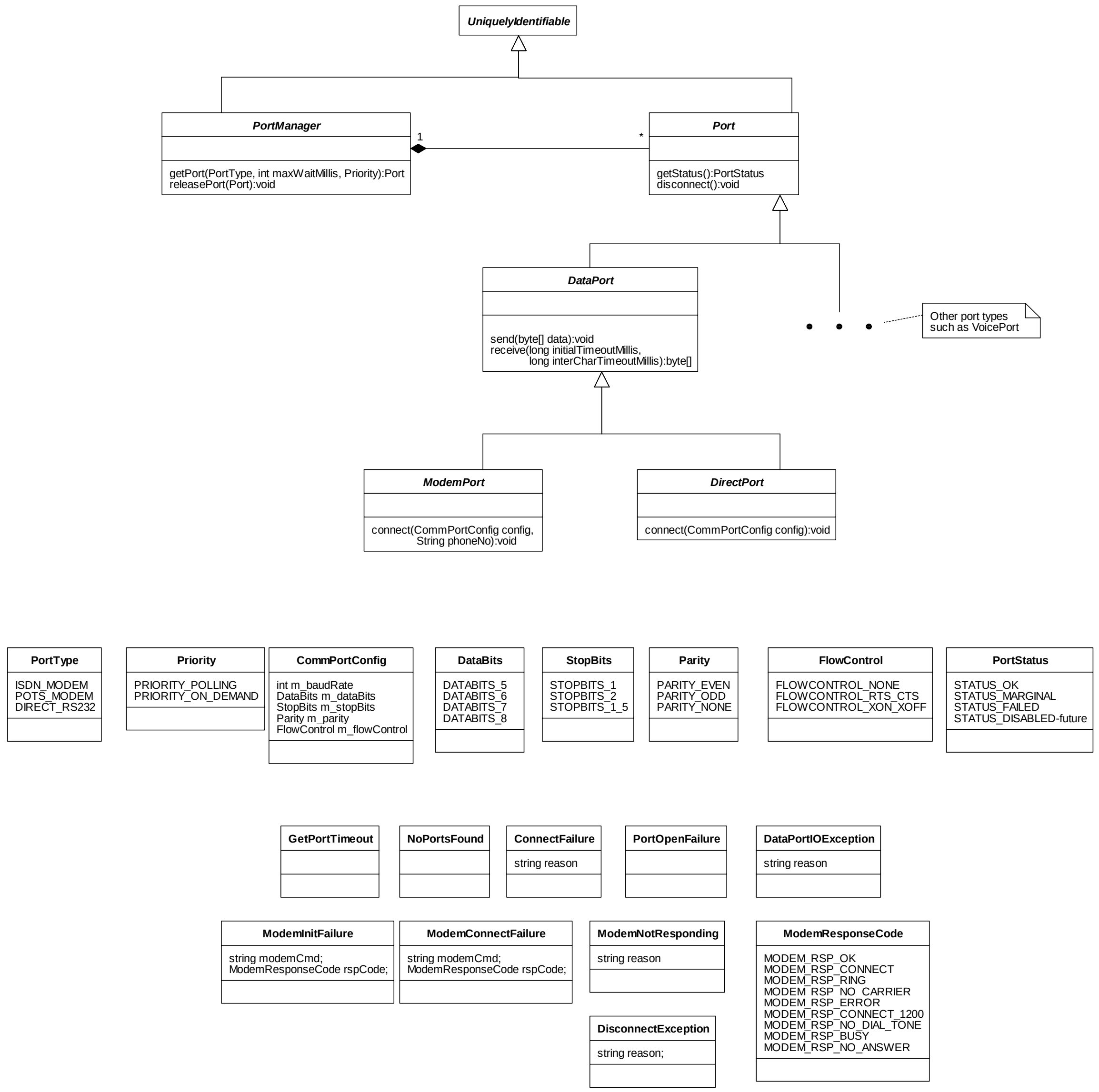


Figure 3. FieldCommunications (Class Diagram)

3.1.1 CommPortConfig (Class)

This structure is used to pass COM port configuration values during a connection request.

3.1.2 ConnectFailure (Class)

This exception is a catchall for exceptions that do not fit in a more specific exception that can be thrown during a connection attempt.

3.1.3 DataBits (Class)

This enumeration defines the valid values for data bits that may be set in a CommPortConfig structure.

3.1.4 DisconnectException (Class)

This exception is thrown when an error is encountered while disconnecting. There is no action that can be taken by the catch handler for this exception except to warn the user. The port will be closed and should be released as normal even if this exception is caught.

3.1.5 FlowControl (Class)

This enumeration defines the valid types of flow control that may be set in a CommPortConfig structure.

3.1.6 DataPort (Class)

A DataPort is a port that allows binary data to be sent and received. Ports of this type support a receive method that allows a chunk of all available data to be received. This method prevents callers from having to issue many receive calls to parse a device response. Instead, this receive call returns all available data received within the timeout parameters. The caller can then parse the data within a local buffer. Using this mechanism, device command and response should require only one call to send and one call to receive.

3.1.7 GetPortTimeout (Class)

This class is an exception that is thrown by a PortManager when a request to acquire a port of a given type cannot be fulfilled within the timeout specified.

3.1.8 ModemPort (Class)

A ModemPort is a communications port that is capable of connecting to a remote modem. ISDN and POTS modems can be implemented under this interface.

3.1.9 NoPortsFound (Class)

This exception is thrown when a port is requested from a PortManager that does not have any of the requested type of port (available or in-use).

3.1.10 Port (Class)

A Port is an object that models a physical communications resource. Derived interfaces specify various types of ports. All ports must be able to supply their status when requested.

3.1.11 Parity (Class)

This enumeration defines the valid values for parity that may be set in a CommPortConfig structure.

3.1.12 PortManager (Class)

A PortManager is an object that manages shared access to communications port resources. The getPort method is used to request the use of a port from the PortManager. Requests for ports specify the type of port needed, the priority of the request, and the maximum time the requester is willing to wait if a port is not immediately available. When the port manager returns a port, the requester has exclusive use of the port until the requester releases the port back to the PortManager or the PortManager reclaims the port due to inactivity.

3.1.13 UniquelyIdentifiable (Class)

This interface will be implemented by all classes that are to be identifiable within the system. The identifier must be generated by the IdentifierGenerator to ensure uniqueness.

3.1.14 PortType (Class)

This enumeration defines the types of ports that may be requested from a PortManager.

3.1.15 Priority (Class)

This enumeration specifies the priority levels used when requesting a port from a PortManager. ON_DEMAND is given higher priority than POLLING.

3.1.16 DirectPort (Class)

A DirectPort is a Port that is directly connected to the target of communications. The connect call needs only to open the communications port.

3.1.17 StopBits (Class)

This enumeration defines the valid values for stop bits that may be set in a CommPortConfig structure.

3.1.18 DataPortIOException (Class)

This exception is used to indicate an Input/Output error has occurred.

3.1.19 ModemInitFailure (Class)

This exception is thrown when there is an error initializing the modem during a connection attempt on a ModemPort.

3.1.20 ModemConnectFailure (Class)

This exception is thrown when there is an error establishing a remote connection via a modem during a connection attempt on a ModemPort. This exception is generated when there is an unfavorable result to the ATDT command on the modem.

3.1.21 ModemNotResponding (Class)

This exception is thrown when there is a failure to command a modem because the modem is not responding to commands.

3.1.22 ModemResponseCode (Class)

This enum defines the result codes for a standard modem.

3.1.23 PortOpenFailure (Class)

This exception is thrown if there is an error opening the port while attempting a connection. This exception would most likely only occur if there is another application accessing the physical com port, which would be true if debugging activities were being done on a port while the FieldCommunications service is still running.

3.1.24 PortStatus (Class)

This enumeration specifies the values used to represent a Port’s status. OK signifies the port is working properly. MARGINAL signifies errors have been experienced during recent use of the port. FAILED indicates the port is not working at all.

3.2 Field Communications Module

3.2.1 Classes

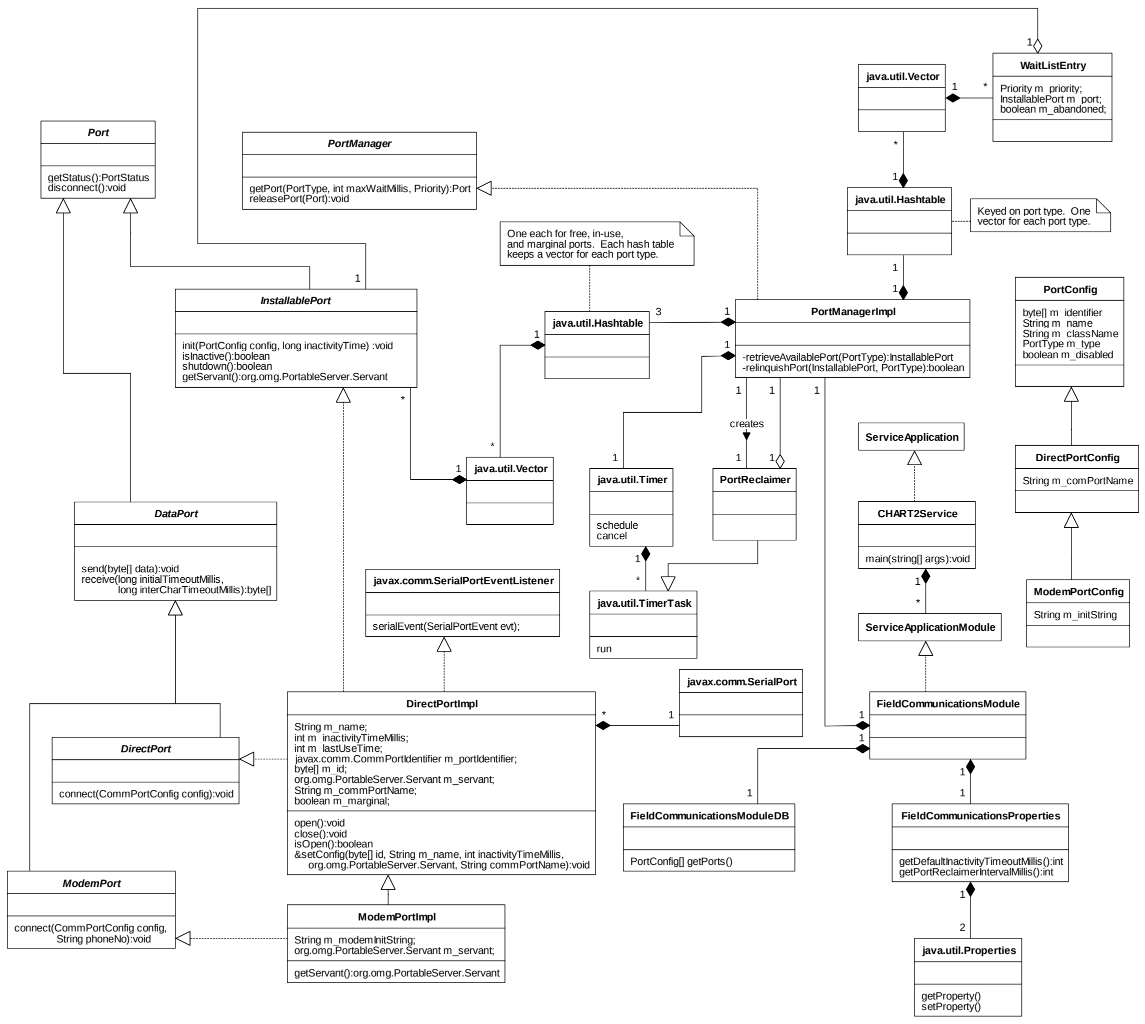

WaitListEntry
Priority m_priority;
InstallablePort m_port;
boolean m_abandoned;
java.util.Vector
java.util.Hashtable
Keyed on port type. One vector for each port type.
Port
getStatus():PortStatus
disconnect():void
PortManager
getPort(PortType, int maxWaitMillis, Priority):Port
releasePort(Port):void
One each for free, in-use, and marginal ports. Each hash table keeps a vector for each port type.
InstallablePort
init(PortConfig config, long inactivityTime) :void
isInactive():boolean
shutdown():boolean
getServant():org.omg.PortableServer.Servant
java.util.Hashtable
PortManagerImpl
-retrieveAvailablePort(PortType):InstallablePort
-relinquishPort(InstallablePort, PortType):boolean
PortConfig
byte[] m_identifier
String m_name
String m_className
PortType m_type
boolean m_disabled
creates
ServiceApplication
java.util.Vector
java.util.Timer
schedule
cancel
PortReclaimer
CHART2Service
main(string[] args):void
DirectPortConfig
String m_comPortName
DataPort
send(byte[] data):void
receive(long initialTimeoutMillis, long interCharTimeoutMillis):byte[]
javax.comm.SerialPortEventListener
serialEvent(SerialPortEvent evt);
java.util.TimerTask
run
ServiceApplicationModule
ModemPortConfig
String m_initString
javax.comm.SerialPort
DirectPortImpl
String m_name;
int m_inactivityTimeMillis;
int m_lastUseTime;
javax.comm.CommPortIdentifier m_portIdentifier;
byte[] m_id;
org.omg.PortableServer.Servant m_servant;
String m_commPortName;
boolean m_marginal;
open():void
close():void
isOpen():boolean
&setConfig(byte[] id, String m_name, int inactivityTimeMillis, org.omg.PortableServer.Servant, String commPortName):void
FieldCommunicationsModule
DirectPort
connect(CommPortConfig config):void
FieldCommunicationsModuleDB
PortConfig[] getPorts()
FieldCommunicationsProperties
getDefaultInactivityTimeoutMillis():int
getPortReclaimerIntervalMillis():int
ModemPort
connect(CommPortConfig config, String phoneNo):void
ModemPortImpl
String m_modemInitString;
org.omg.PortableServer.Servant m_servant;
getServant():org.omg.PortableServer.Servant
java.util.Properties
getProperty()
setProperty()

Figure 4. FieldCommunicationsModule (Class Diagram)

3.2.1.1 CHART2Service (Class)

The CHART2Service is an application that helps in installation and termination of the modules in CHART II system.

3.2.1.2 DataPort (Class)

A DataPort is a port that allows binary data to be sent and received. Ports of this type support a receive method that allows a chunk of all available data to be received. This method prevents callers from having to issue many receive calls to parse a device response. Instead, this receive call returns all available data received within the timeout parameters. The caller can then parse the data within a local buffer. Using this mechanism, device command and response should require only one call to send and one call to receive.

3.2.1.3 DirectPort (Class)

A DirectPort is a Port that is directly connected to the target of communications. The connect call needs only to open the communications port.

3.2.1.4 DirectPortConfig (Class)

This class holds configuration data for a direct connect port, which includes only the name of the COM port.

3.2.1.5 DirectPortImpl (Class)

This class implements the DirectPort interface as defined in the IDL. Its connect method opens a javax.comm.SerialPort object and sets the port settings according to the baud, data bits, stop bits, and parity that was passed. Its disconnect method closes the javax.comm.SerialPort. This class also implements the send and receive functions as specified in the DataPort IDL interface. The send and receive methods use the read and write methods of the javax.comm.SerialPort object to send and receive bytes on the COM port. While the send method contains little processing other than calling the javax.comm.SerialPort object's write method, the receive method contains logic that allows it to receive a burst of bytes before returning. This causes the receive method to return all available bytes on the port and thus helps to prevent the need for multiple calls to receive for a single command response. This class updates a timestamp each time send or receive is called. When its isInactive() method is called, it checks the current time vs. the last send/receive time and if the difference is greater than the current inactivity timeout, it returns true.

3.2.1.6 FieldCommunicationsModule (Class)

This class is a service application module that can be installed into a CHART2Service. This module serves one PortManager object that provides access to one or more Port objects. It publishes the reference to this PortManager in the CORBA Trader. This class contains a FieldCommunicationsModuleDB object used to provide database access to the other classes within the package.

3.2.1.7 InstallablePort (Class)

This interface is implemented by Port implementations that can be installed into the FieldCommunicationsModule and PortManager generically. The PortManagerImpl instantiates the specific impl using the class name that is part of a port's configuration data. The PortManager then calls each port's init method to allow each port to initialize its internal state. The PortManagerImpl's use of this interface allows it to manage all types of ports (current and future) in a generic way.

3.2.1.8 ModemPortConfig (Class)

This class holds configuration data that is specific to modem ports. The COM port name is included as well as the type of modem port (ISDN or POTS) and a default modem initialization string.

3.2.1.9 PortConfig (Class)

This class holds data that is common to all types of ports. The PortManager uses this data to generically construct port objects.

3.2.1.10 FieldCommunicationsModuleDB (Class)

This class provides methods used access Field Communications configuration data. The getPorts() method returns an array of PortConfig derived objects that contain configuration data specific to the type of port that has been configured. The configuration data is retrieved from a configuration file where PortConfig objects were previously persisted.

3.2.1.11 java.util.Vector (Class)

The Vector class implements a *growable* array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created. (extracted from the JDK 1.3 javadoc)

3.2.1.12 ModemPort (Class)

A ModemPort is a communications port that is capable of connecting to a remote modem. ISDN and POTS modems can be implemented under this interface.

3.2.1.13 Port (Class)

A Port is an object that models a physical communications resource. Derived interfaces specify various types of ports. All ports must be able to supply their status when requested.

3.2.1.14 PortManagerImpl (Class)

This class implements the PortManager interface as specified in the IDL. Hashtables are used to keep lists of ports according to their port type. Three of these hashtables are used to separate ports based on their current state—in use, available, or marginal. Ports that are in the marginal hashtable are available but are in a marginal state. The getPort method looks for an available port in the available list prior to the marginal list.

3.2.1.15 ServiceApplicationModule (Class)

This interface is implemented by modules that serve CORBA objects. Implementing classes are notified when their host service is initialized and when it is shutdown. The implementing class can use these notifications along with the services provided by the invoking ServiceApplication to perform actions such as object creation and publication.

3.2.1.16 java.util.Hashtable (Class)

This class implements a hashtable, which is a data structure that maps keys to values. Any non-null object can be used as a key or as a value. Objects used as keys implement the hashCode method that is inherited by all objects from the java.lang.Object class.

3.2.1.17 javax.comm.SerialPortEventListener (Class)

This interface is implemented by objects that wish to be notified of events that occur on a javax.comm.SerialPort.

3.2.1.18 FieldCommunicationsProperties (Class)

This class provides access to properties in the Chart2Service properties file that are specific to the FieldCommunicationsModule.

3.2.1.19 java.util.Properties (Class)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. A property list can contain another property list as its “defaults”; this second property list is searched if the property key is not found in the original property list.

3.2.1.20 java.util.Timer (Class)

This class provides asynchronous execution of tasks that are scheduled for one-time or recurring execution.

3.2.1.21 PortReclaimer (Class)

This class is a timer task that is scheduled to run periodically and cause the PortManager to determine if any in-use ports have had excessive idle time. When the PortManager discovers ports that are in-use but have not had activity within a configurable time period, the port manager disconnects the object, deactivates the object in the POA, and puts the port back in the free list.

3.2.1.22 java.util.TimerTask (Class)

This class is an abstract base class which can be scheduled with a timer to be executed one or more times.

3.2.1.23 javax.comm.SerialPort (Class)

This class provides access to a computer’s serial port. It allows the port to be opened and closed and allows data to be sent and received.

3.2.1.24 ModemPortImpl (Class)

This class implements the ModemPort interface as defined in IDL. The ModemPortImpl’s connect method calls its base class connect method that opens a communications port. The connect method then goes on to initialize and dial the modem and determine if the modem has connected to a remote modem. The disconnect method interrupts the modem, hangs up the modem, and calls the base class disconnect method which closes the COM port. This class inherits its base class (DirectPortImpl) send and receive methods which send and receive data over the connected modem.

3.2.1.25 PortManager (Class)

A PortManager is an object that manages shared access to communications port resources. The getPort method is used to request the use of a port from the PortManager. Requests for ports specify the type of port needed, the priority of the request, and the maximum time the requester is willing to wait if a port is not immediately available. When the port manager returns a port, the requester has exclusive use of the port until the requester releases the port back to the PortManager or the PortManager reclaims the port due to inactivity.

3.2.1.26 ServiceApplication (Class)

This interface is implemented by objects that can provide the basic services needed by a CHART II service application. These services include providing access to basic CORBA objects that are needed by service applications, such as the ORB, POA, Trader, and Event Service.

3.2.1.27 WaitListEntry (Class)

This class contains values that are placed on a wait list to allow prioritized fulfillment of requests for ports.

3.2.2 Sequence Diagrams

3.2.2.1 DirectPortImpl:close (Sequence Diagram)

A DirectPortImpl processes a close request by delegating the call to the javax.comm.SerialPort object and then setting associated member variables to null.

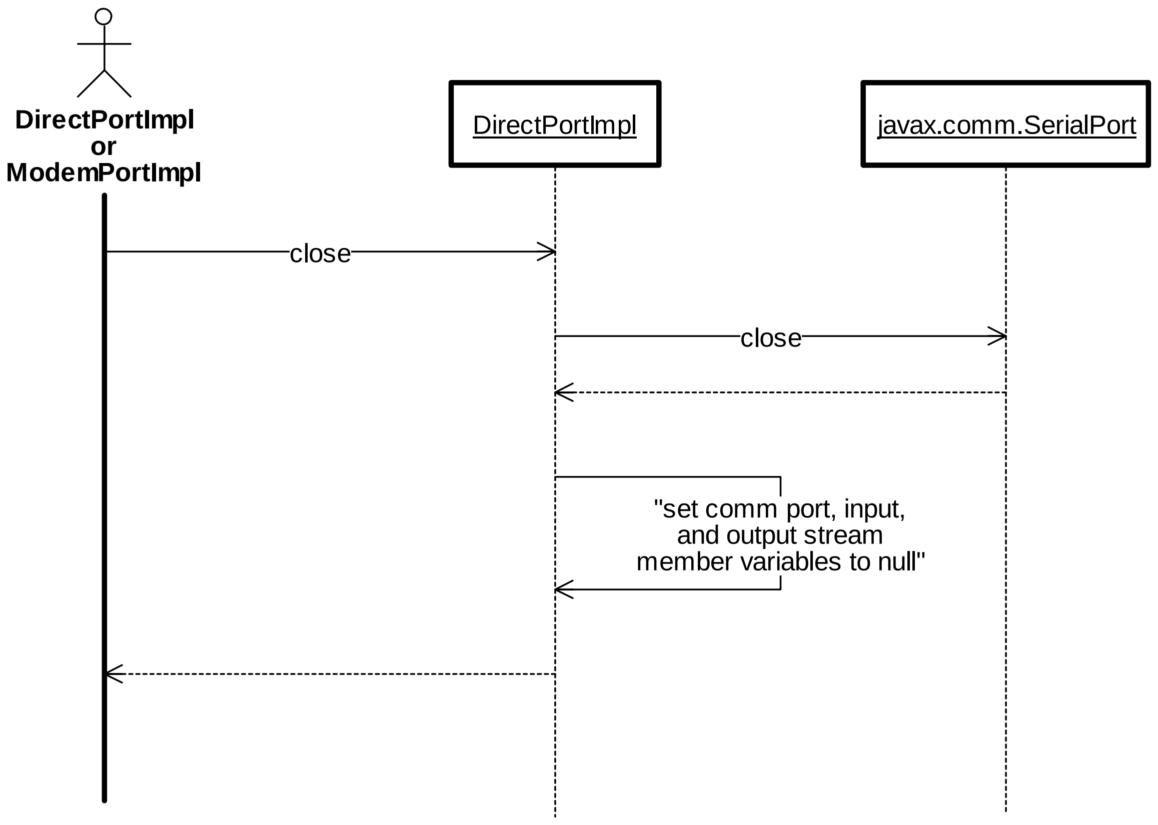


Figure 5. DirectPortImpl:close (Sequence Diagram)

3.2.2.2 DirectPortImpl:Connect (Sequence Diagram)

The DirectPortImpl processes a connect request by first calling its open method (See DirectPortImpl:open) and then setting the serial port settings according to the parameters passed by the caller.

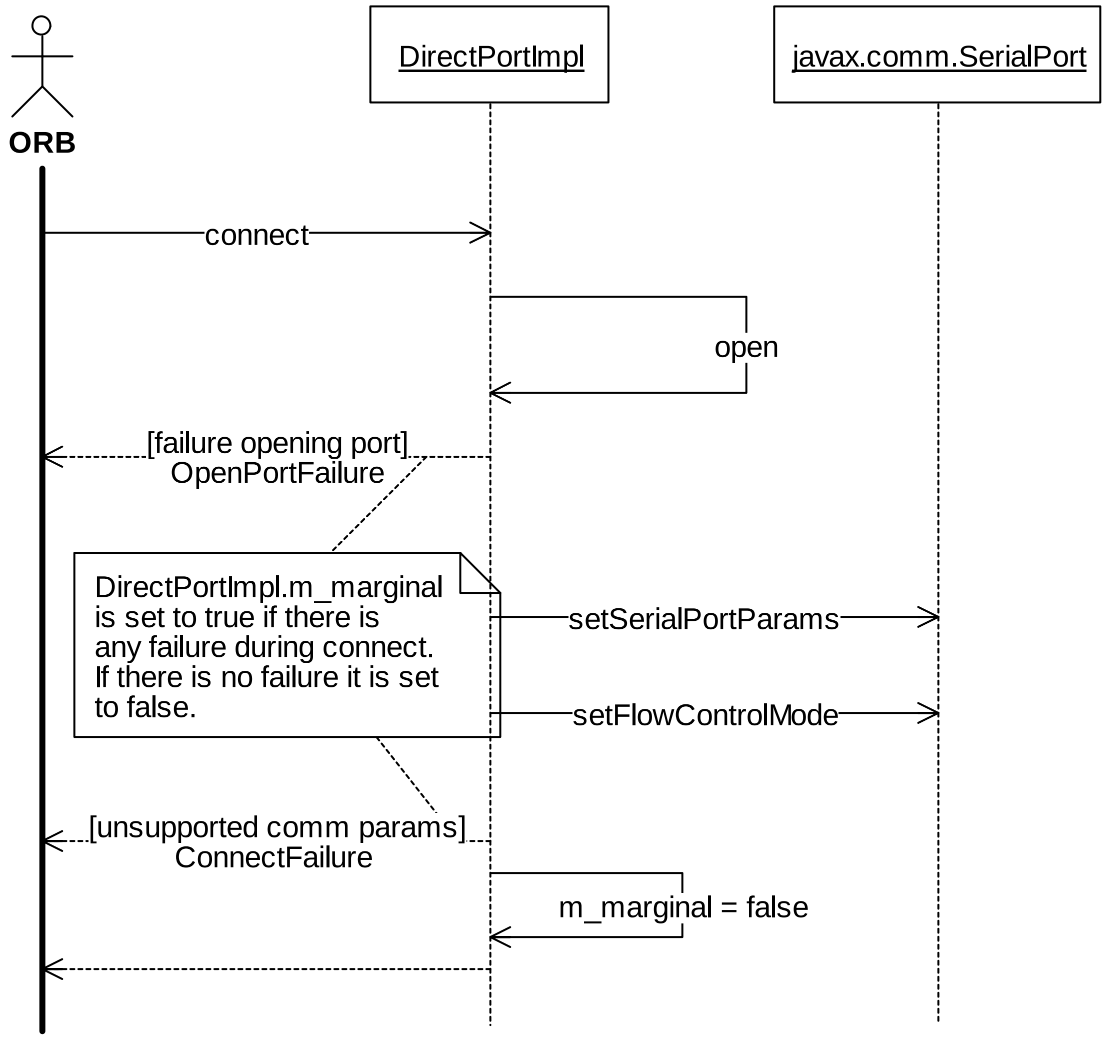

DirectPortImpl
javax.comm.SerialPort
ORB
connect
open
[failure opening port] OpenPortFailure
DirectPortImpl.m_marginal is set to true if there is any failure during connect. If there is no failure it is set to false.
setSerialPortParams
setFlowControlMode
[unsupported comm params] ConnectFailure
m_marginal = false

Figure 6. DirectPortImpl:Connect (Sequence Diagram)

3.2.2.3 DirectPortImpl:disconnect (Sequence Diagram)

The DirectPortImpl processes the disconnect request by calling its own close method. If disconnect is called on a port that is already disconnected, the method simply returns fast and no exception is thrown.

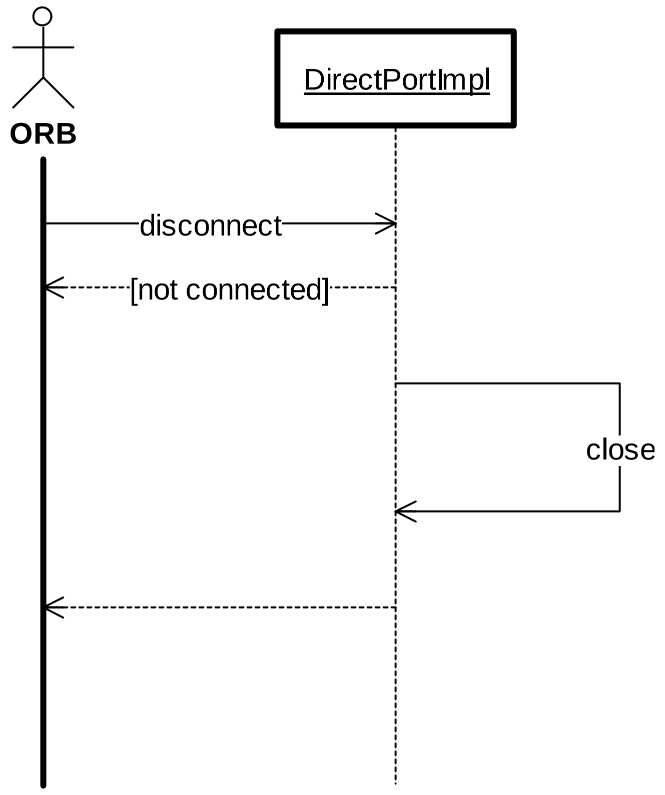


Figure 7. DirectPortImpl:disconnect (Sequence Diagram)

3.2.2.4 DirectPortImpl:init (Sequence Diagram)

When a DirectPortImpl is initialized by the PortManagerImpl, it retrieves information specific to this port type from the database, which in this case is only the COM port name this object provides access to. A CommPortIdentifier is retrieved using the specified COM port name. If the COM port name given is not an existing serial port on the machine where the DirectPortImpl is running, an exception is thrown.

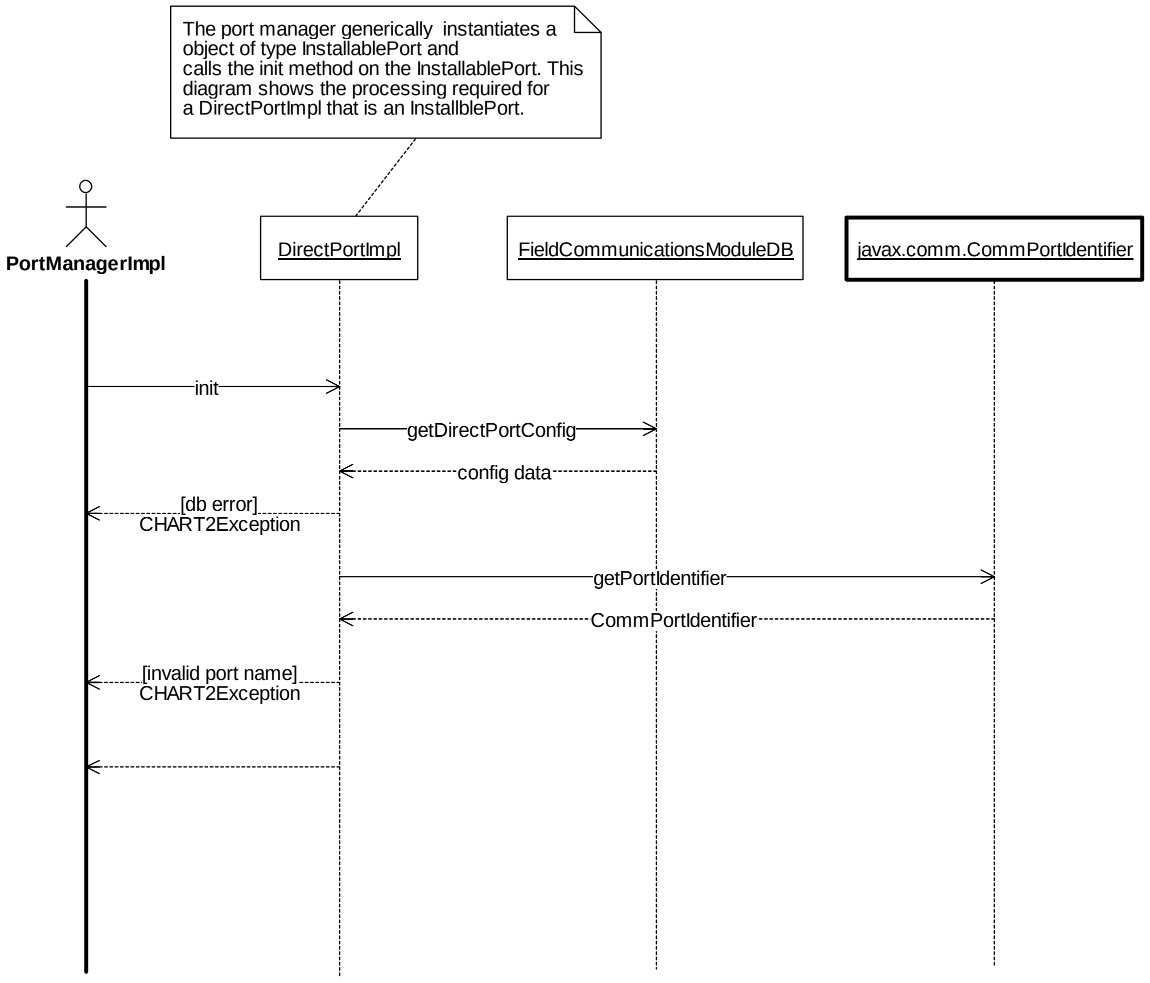

The port manager generically instantiates a object of type InstallablePort and calls the init method on the InstallablePort. This diagram shows the processing required for a DirectPortImpl that is an InstallblePort.
PortManagerImpl
DirectPortImpl
FieldCommunicationsModuleDB
javax.comm.CommPortIdentifier
init
getDirectPortConfig
config data
[db error] CHART2Exception
getPortIdentifier
CommPortIdentifier
[invalid port name] CHART2Exception

Figure 8. DirectPortImpl:init (Sequence Diagram)

3.2.2.5 DirectPortImpl:open (Sequence Diagram)

When a DirectPortImpl's open method is called, it retrieves an instance of a javax.comm.SerialPort from the CommPortIdentifier that was created during initialization. After the SerialPort object is retrieved, its input and output streams are retrieved for later use during send and receive operations. The DirectPortImpl adds itself as an EventListener on the SerialPort and enables events that signify data is available on the port. This asynchronous notification of data being available is used in the receive method's processing. If the port is in use by another application, this method throws a PortOpenFailure exception.

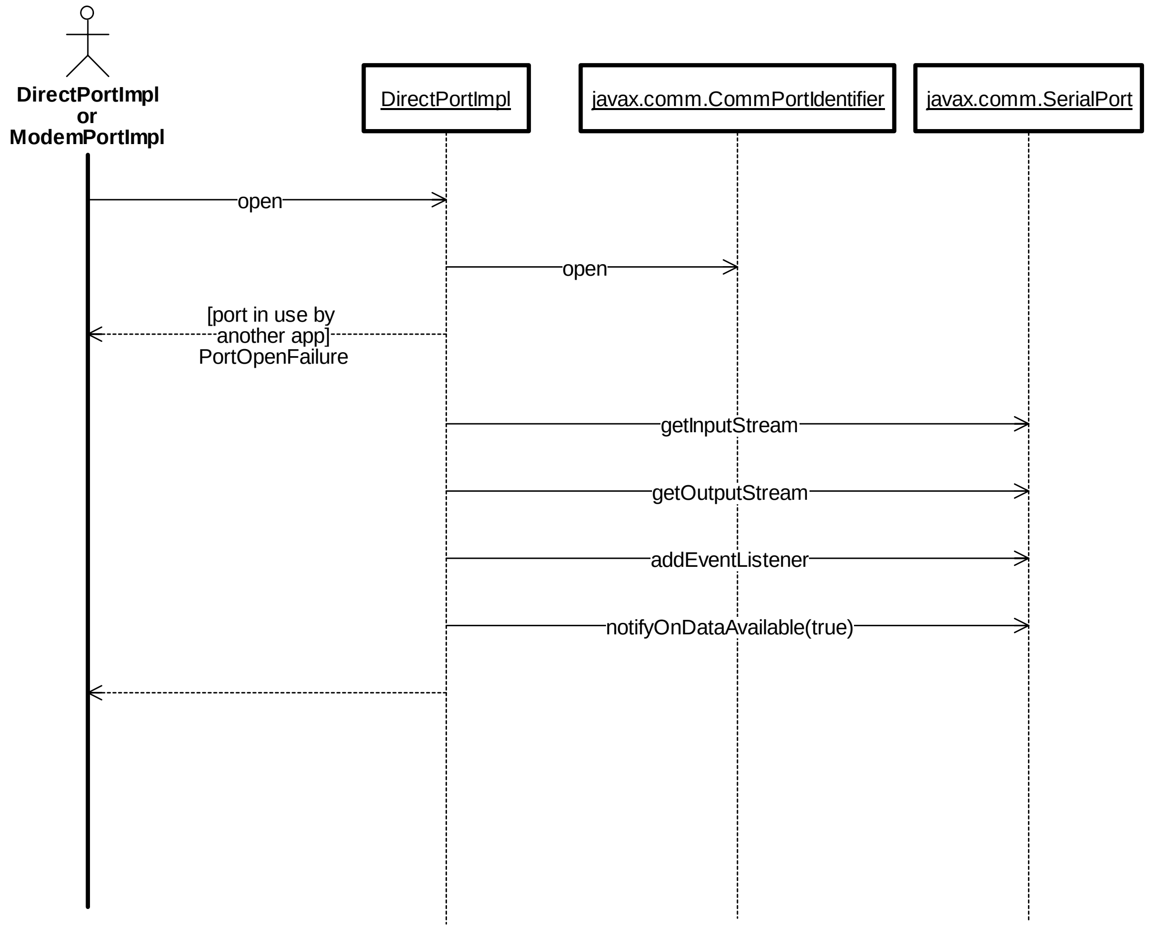

DirectPortImpl or ModemPortImpl
DirectPortImpl
javax.comm.CommPortIdentifier
javax.comm.SerialPort
open
open
[port in use by another app] PortOpenFailure
getInputStream
getOutputStream
addEventListener
notifyOnDataAvailable(true)

Figure 9. DirectPortImpl:open (Sequence Diagram)

3.2.2.6 DirectPortImpl:receive (Sequence Diagram)

The DirectPortImpl receive method is customized for use with command/response devices by returning all bytes in a response burst in a single call to receive. The caller specifies two timeouts, the time to wait for the first byte to arrive and the maximum time to wait to determine that a complete burst of bytes has been received. Using this mechanism, in most cases a single call to this receive method will return the complete device response. In the unlikely event that the entire device response is not received in a single call to receive(), the caller can call receive again to get the remainder of the packet. (Protocol handlers are coded to handle this condition should it arise).

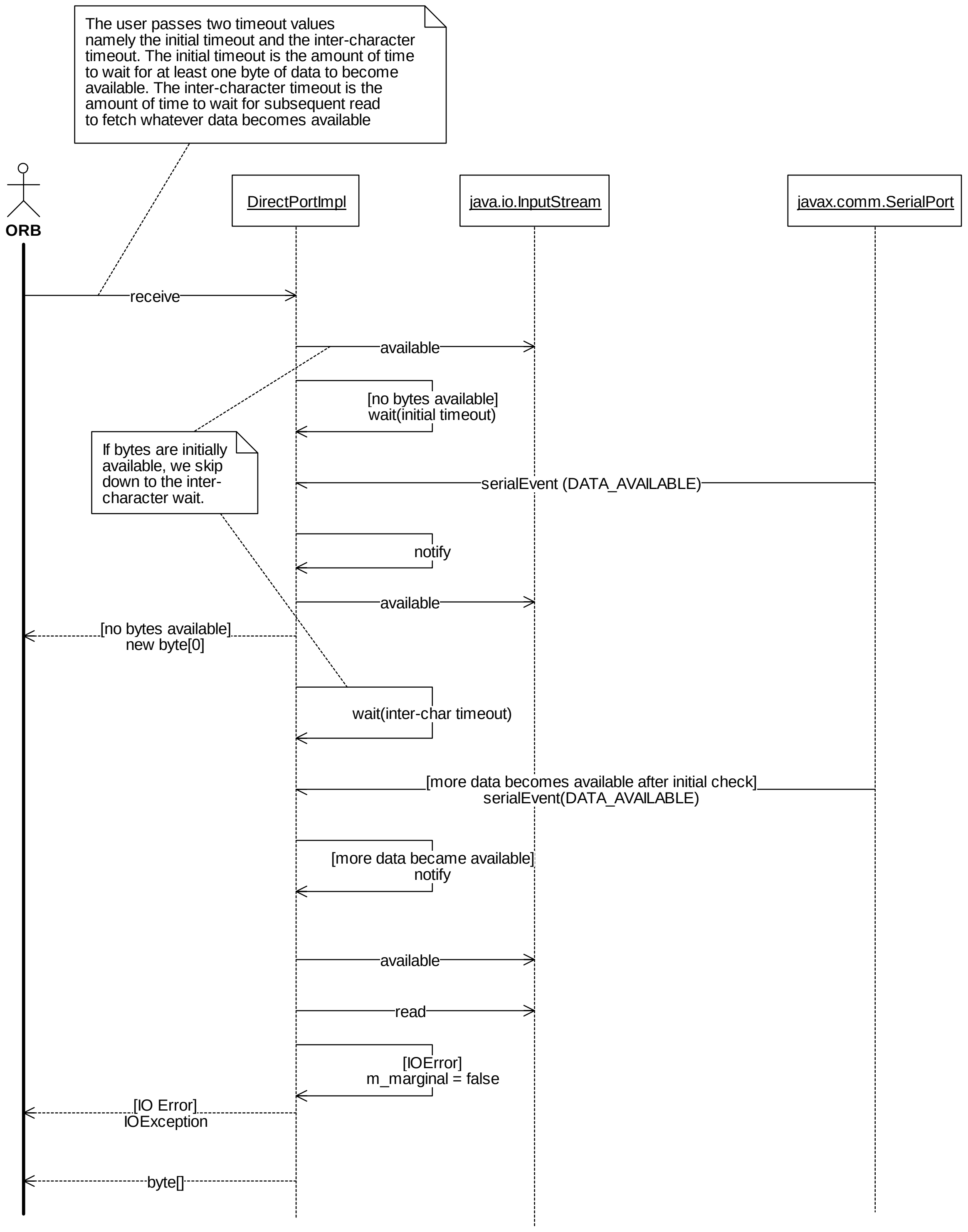

The user passes two timeout values namely the initial timeout and the inter-character timeout. The initial timeout is the amount of time to wait for at least one byte of data to become available. The inter-character timeout is the amount of time to wait for subsequent read to fetch whatever data becomes available
ORB
DirectPortImpl
java.io.InputStream
javax.comm.SerialPort
receive
available
[no bytes available] wait(initial timeout)
If bytes are initially available, we skip down to the inter-character wait.
serialEvent (DATA_AVAILABLE)
notify
available
[no bytes available] new byte[0]
wait(inter-char timeout)
[more data becomes available after initial check] serialEvent(DATA_AVAILABLE)
[more data became available] notify
available
read
[IOError] m_marginal = false
[IO Error] IOException
byte[]

Figure 10. DirectPortImpl:receive (Sequence Diagram)

3.2.2.7 DirectPortImpl:Send (Sequence Diagram)

The DirectPortImpl processes a send request by delegating the request to the output stream of the javax.comm.Serial port object. If a java.io.IOException is thrown by the output stream, the exception is caught and re-thrown as a CORBA exception.

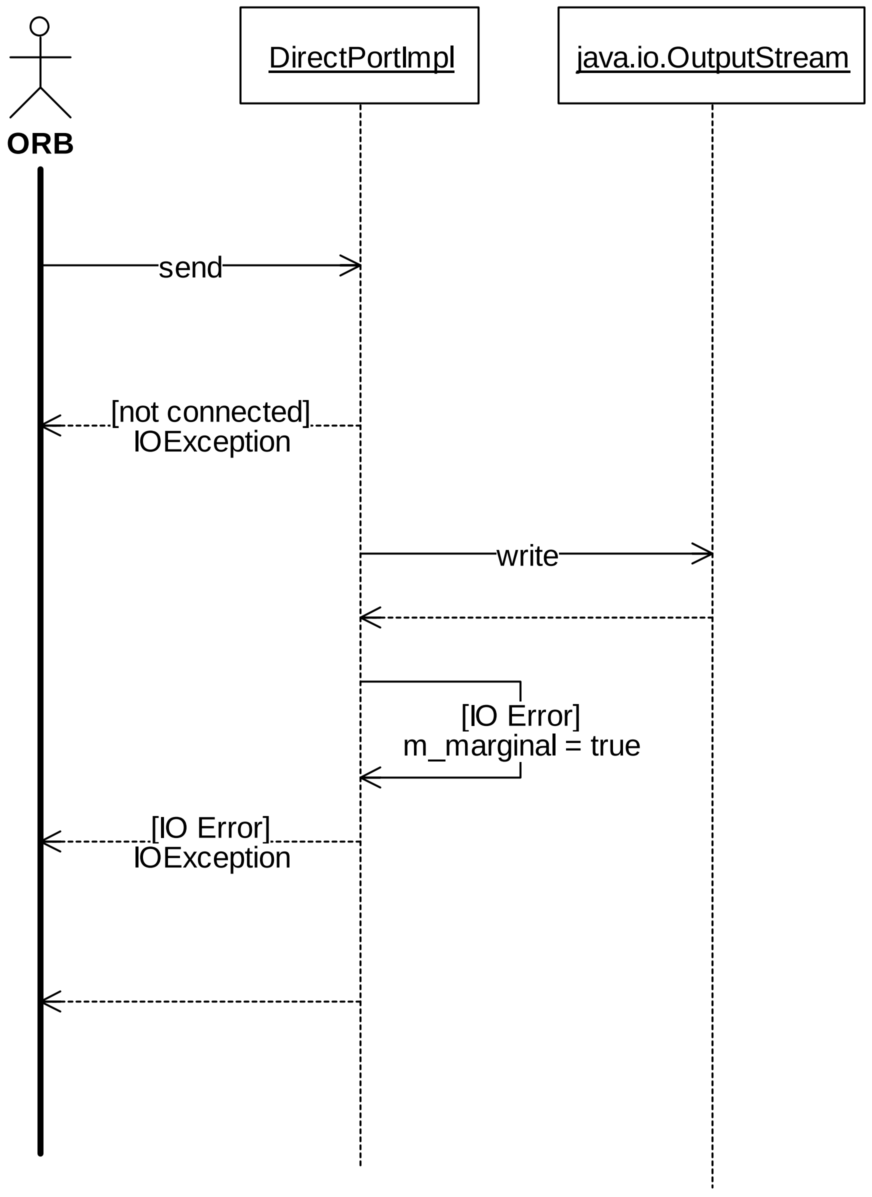


Figure 11. DirectPortImpl:Send (Sequence Diagram)

3.2.2.8 DirectPortImpl:shutdown (Sequence Diagram)

When a DataPortImpl object is shutdown by the PortManagerImpl, the DataPortImpl closes itself if it is currently open.

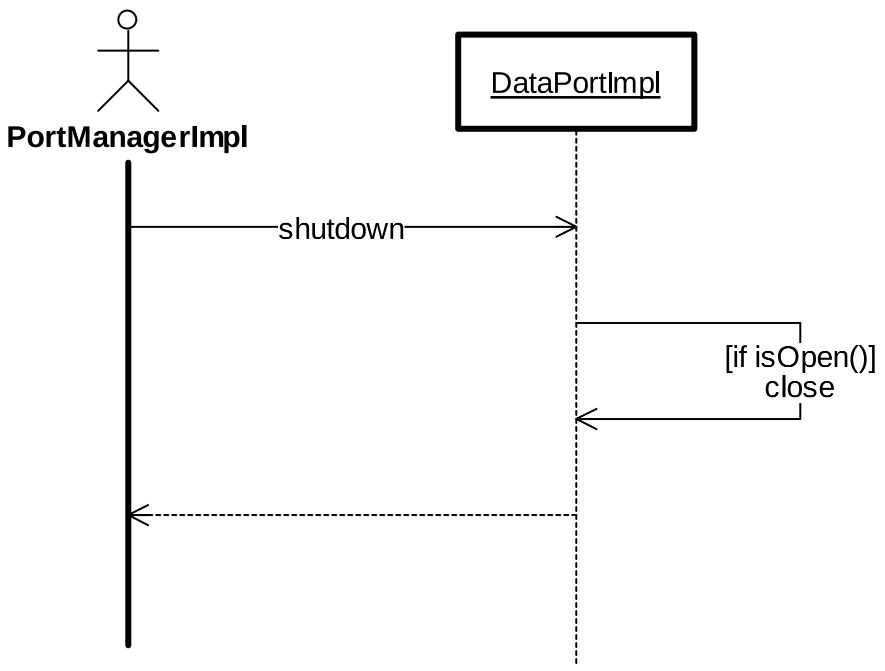


Figure 12. DirectPortImpl:shutdown (Sequence Diagram)

3.2.2.9 FieldCommunicationsModule:initialize (Sequence Diagram)

When the FieldCommunicationsModule is initialized from the Chart2Service, it obtains objects it will need during processing from the Chart2Service via the ServiceApplication interface. The FieldCommunicationsModule constructs a single PortManagerImpl object. The PortManagerImpl creates four Hashtables: three to manage ports and one to manage port requests that are waiting for a port to free up. Each hash table contains a number of vectors, one for each type of port that is being managed by the PortManagerImpl. These Vectors are added as the first port of a given type is encountered, thus after initialization, each Hashtable contains one vector for each type of port being managed by this particular instance of the PortManagerImpl. All synchronization done by the PortManagerImpl is done using the freeList Vector for the specific type of port that is being dealt with, thus getPort() and releasePort() calls for one port type do not synchronize with getPort and releasePort() calls for other port types. The synchronization on the freeList is used to synchronize access to all the other lists, including the wait list, because the getPort() and release() port operations typically manipulate more than one list during their processing. The PortManagerImpl creates a Timer to be used to periodically wake the PortManagerImpl and have it check its inUseList for inactive ports. After the PortManagerImpl has been created the FieldCommunicationsModule activates the object on the persistent POA to keep the object reference for the PortManager consistent across multiple object/server life times. The FieldCommunicationsModule uses the ServiceApplication interface's registerObject method to publish the object in the Trader and to take care of withdrawal from the trader when necessary.

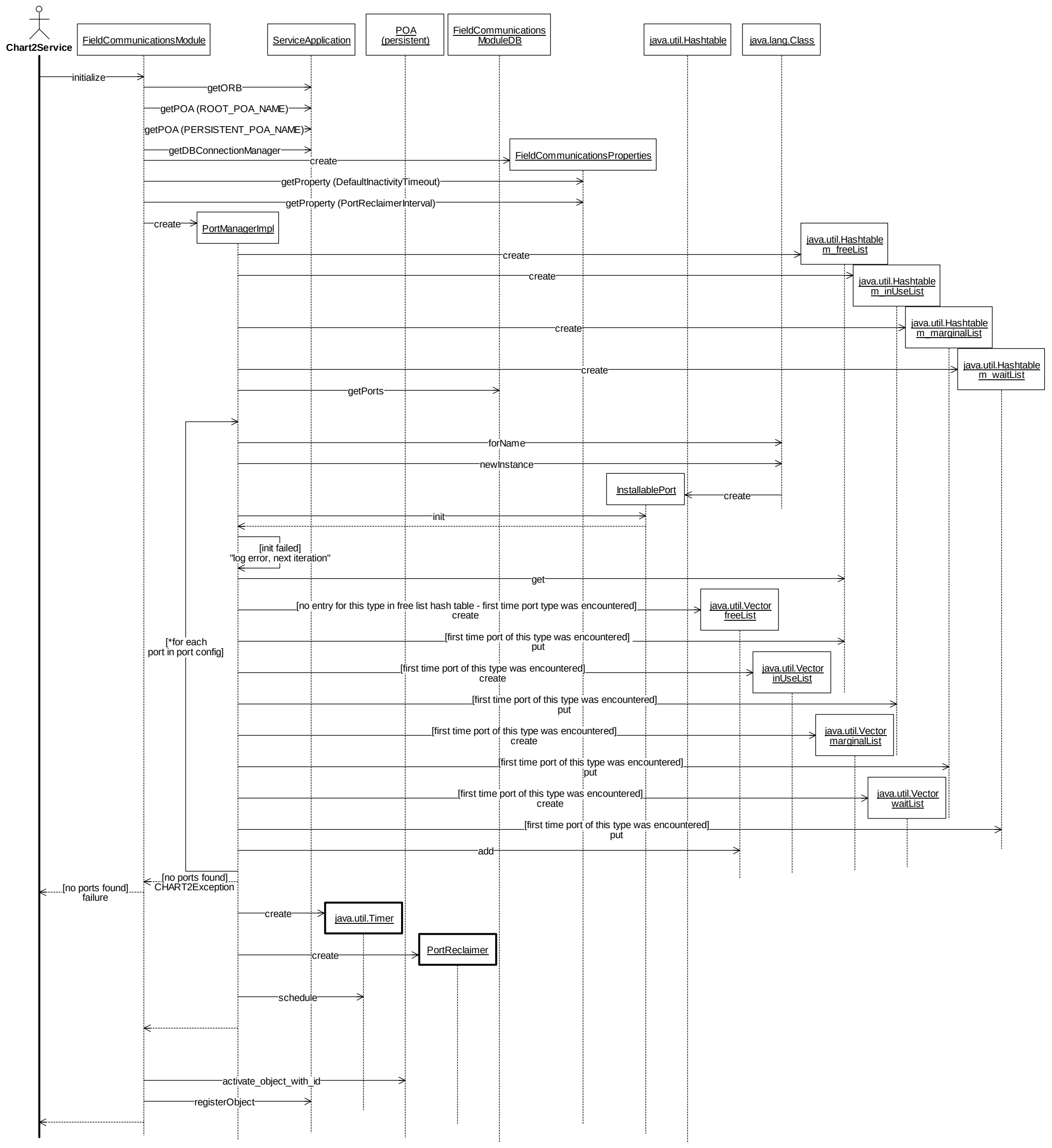


Chart2Service
FieldCommunicationsModule
ServiceApplication
POA (persistent)
FieldCommunications ModuleDB
java.util.Hashtable
java.lang.Class
initialize
getORB
getPOA (ROOT_POA_NAME)
getPOA (PERSISTENT_POA_NAME)
getDBConnectionManager
create
FieldCommunicationsProperties
getProperty (DefaultInactivityTimeout)
getProperty (PortReclaimerInterval)
create
PortManagerImpl
create
java.util.Hashtable m_freeList
create
java.util.Hashtable m_inUseList
create
java.util.Hashtable m_marginalList
create
java.util.Hashtable m_waitList
getPorts
forName
newInstance
InstallablePort
create
init
[init failed] "log error, next iteration"
get
[no entry for this type in free list hash table - first time port type was encountered] create
java.util.Vector freeList
[*for each port in port config]
[first time port of this type was encountered] put
[first time port of this type was encountered] create
java.util.Vector inUseList
[first time port of this type was encountered] put
[first time port of this type was encountered] create
java.util.Vector marginalList
[first time port of this type was encountered] put
[first time port of this type was encountered] create
java.util.Vector waitList
[first time port of this type was encountered] put
add
[no ports found] CHART2Exception
[no ports found] failure
create
java.util.Timer
create
PortReclaimer
schedule
activate_object_with_id
registerObject

Figure 13. FieldCommunicationsModule:initialize (Sequence Diagram)

3.2.2.10 FieldCommunicationsModule:Shutdown (Sequence Diagram)

When the FieldCommunicationsModule is shutdown by the Chart2Service it cancels the timer used to periodically run the ReclaimPorts task. Each list for each port type is then emptied, shutting down each port object that exists. The ports that are in the inUse list are deactivated from the POA prior to being shutdown. The port's shutdown method takes care of disconnecting any port that is currently connected.

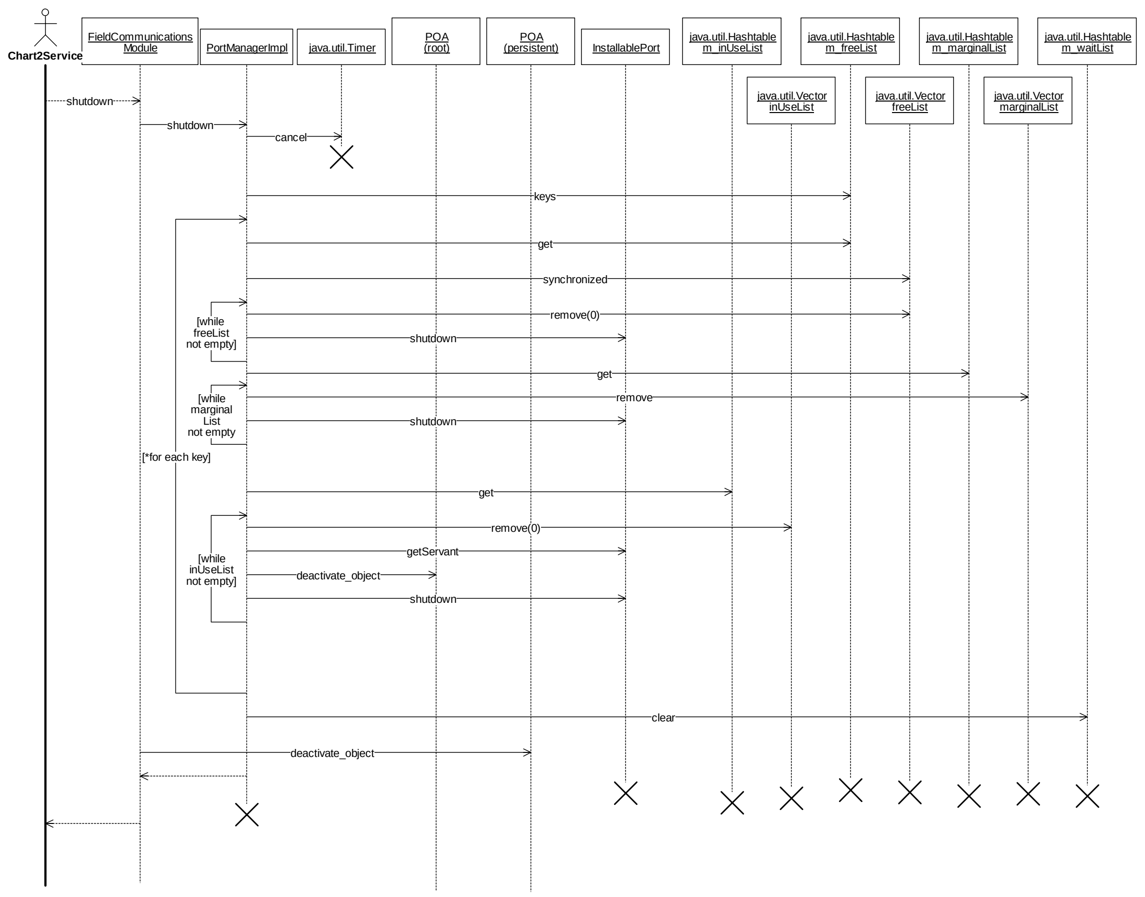


Chart2Service
FieldCommunications Module
PortManagerImpl
java.util.Timer
POA (root)
POA (persistent)
InstallablePort
java.util.Hashtable m_inUseList
java.util.Hashtable m_freeList
java.util.Hashtable m_marginalList
java.util.Hashtable m_waitList
java.util.Vector inUseList
java.util.Vector freeList
java.util.Vector marginalList
shutdown
shutdown
cancel
keys
get
synchronized
[while freeList not empty]
remove(0)
shutdown
get
[while marginal List not empty
remove
shutdown
[*for each key]
get
[while inUseList not empty]
remove(0)
getServant
deactivate_object
shutdown
clear
deactivate_object

Figure 14. FieldCommunicationsModule:Shutdown (Sequence Diagram)

3.2.2.11 ModemPortImpl:Connect (Sequence Diagram)

A ModemPortImpl processes a connect request by first calling its base class (DirectPortImpl) connect method. This opens the communications port and readies it for send and receive calls. The ModemPortImpl then calls the base class send and receive methods to send modem commands to the modem, first to initialize the modem and then to dial the modem. The ModemPortImpl parses the modem responses and passes a detailed exception should any problems occur.

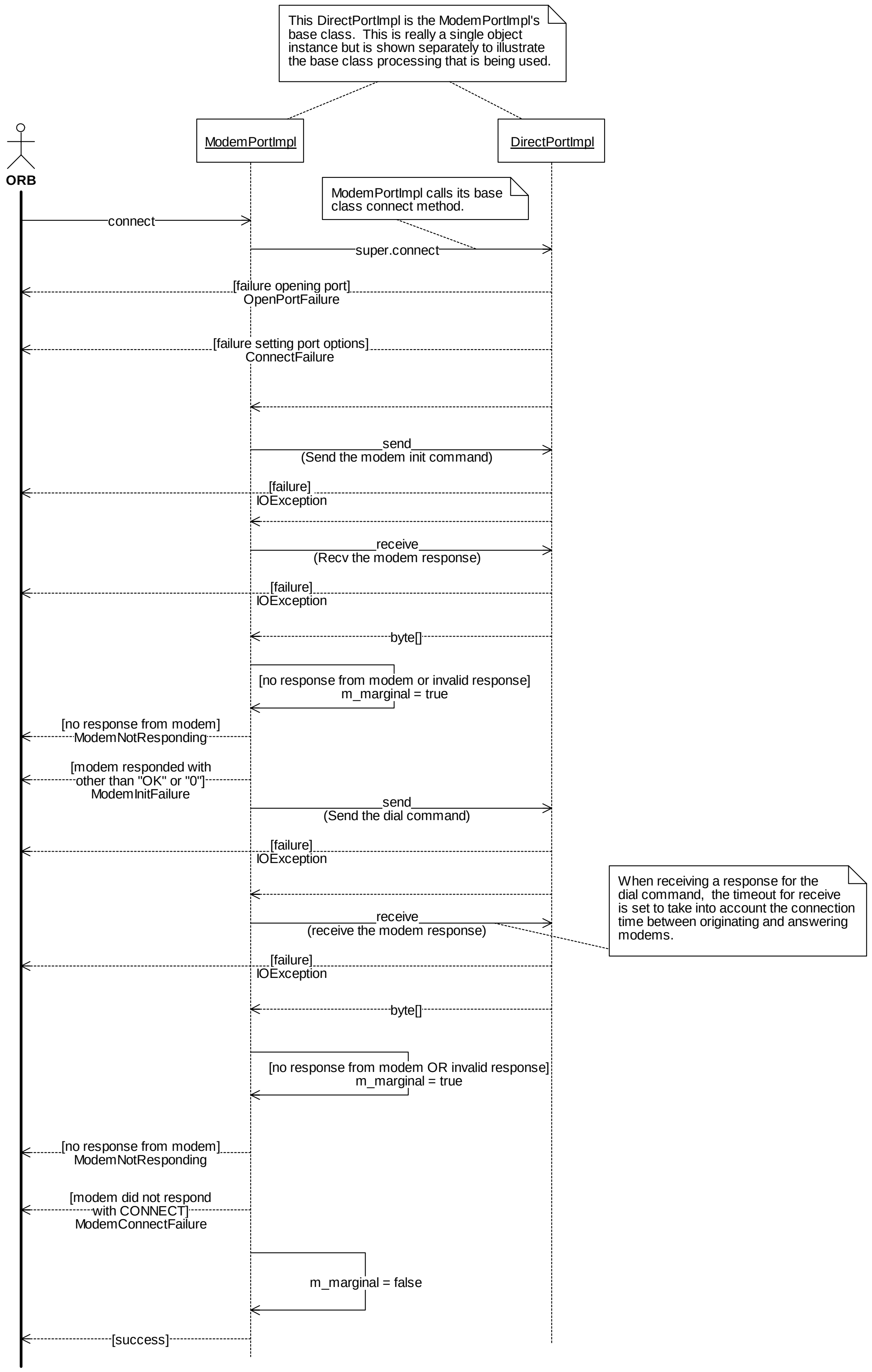

This DirectPortImpl is the ModemPortImpl's base class. This is really a single object instance but is shown separately to illustrate the base class processing that is being used.
ModemPortImpl
DirectPortImpl
ORB
ModemPortImpl calls its base class connect method.
connect
super.connect
[failure opening port] OpenPortFailure
[failure setting port options] ConnectFailure
send (Send the modem init command)
[failure] IOException
receive (Recv the modem response)
[failure] IOException
byte[]
[no response from modem or invalid response] m_marginal = true
[no response from modem] ModemNotResponding
[modem responded with other than "OK" or "0"] ModemInitFailure
send (Send the dial command)
[failure] IOException
When receiving a response for the dial command, the timeout for receive is set to take into account the connection time between originating and answering modems.
receive (receive the modem response)
[failure] IOException
byte[]
[no response from modem OR invalid response] m_marginal = true
[no response from modem] ModemNotResponding
[modem did not respond with CONNECT] ModemConnectFailure
m_marginal = false
[success]

Figure 15. ModemPortImpl:Connect (Sequence Diagram)

3.2.2.12 ModemPortImpl:disconnect (Sequence Diagram)

When the ModemPortImpl processes a disconnect request, it uses its base class (DirectPortImpl) send and receive methods to command the modem to hang up. Before issuing the hangup command the +++ command must be issued to the modem to put the modem back into command mode. One second of inactivity must exist prior to and after the +++ command to interrupt the modem. After hanging up the modem, the ModemPortImpl calls the base class close method to close the serial port.

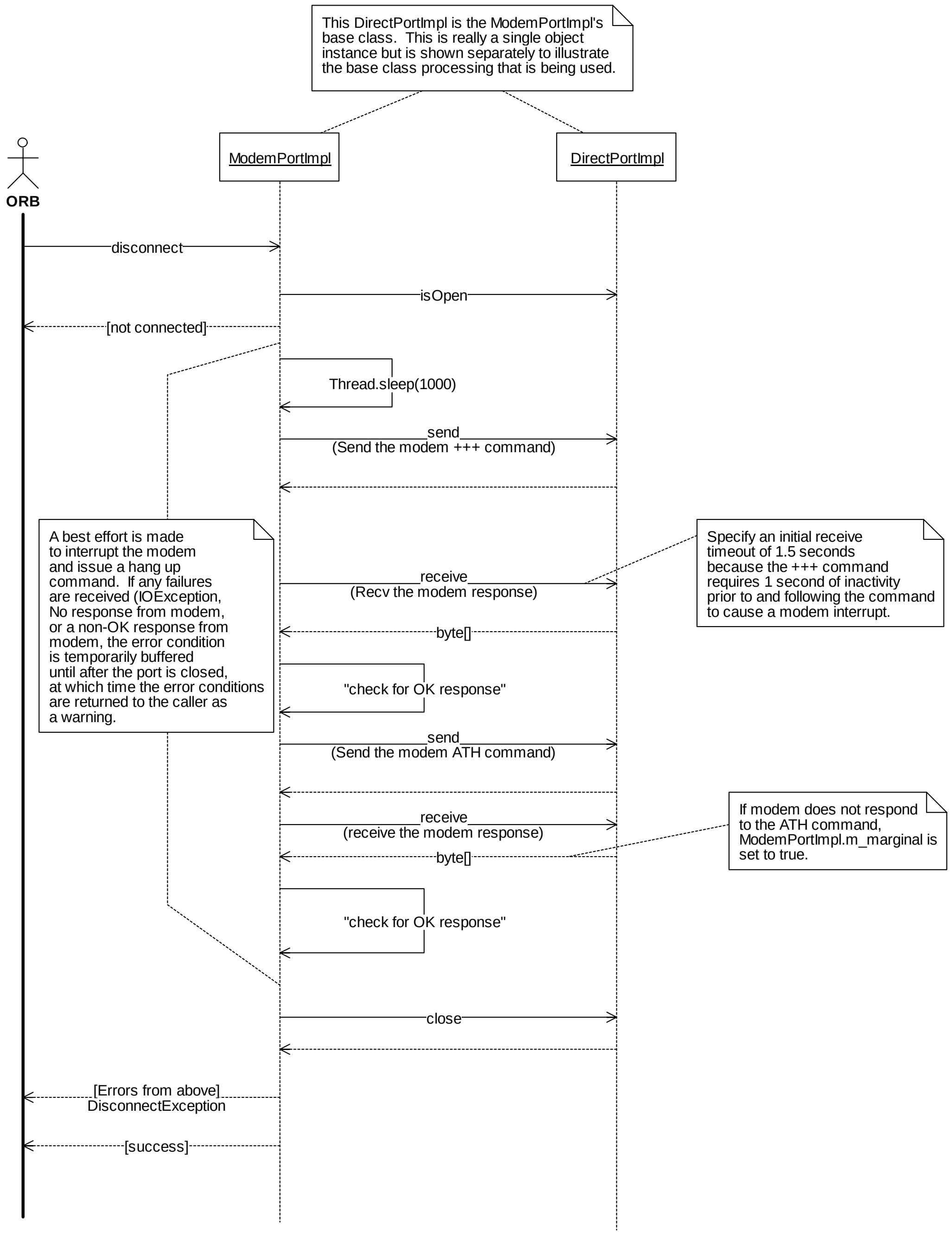

This DirectPortImpl is the ModemPortImpl's base class. This is really a single object instance but is shown separately to illustrate the base class processing that is being used.
ORB
ModemPortImpl
DirectPortImpl
disconnect
isOpen
[not connected]
Thread.sleep(1000)
send
(Send the modem +++ command)
A best effort is made to interrupt the modem and issue a hang up command. If any failures are received (IOException, No response from modem, or a non-OK response from modem, the error condition is temporarily buffered until after the port is closed, at which time the error conditions are returned to the caller as a warning.
receive
(Recv the modem response)
Specify an initial receive timeout of 1.5 seconds because the +++ command requires 1 second of inactivity prior to and following the command to cause a modem interrupt.
byte[]
"check for OK response"
send
(Send the modem ATH command)
receive
(receive the modem response)
If modem does not respond to the ATH command, ModemPortImpl.m_marginal is set to true.
byte[]
"check for OK response"
close
[Errors from above]
DisconnectException
[success]

Figure 16. ModemPortImpl:disconnect (Sequence Diagram)

3.2.2.13 ModemPortImpl:init (Sequence Diagram)

When a ModemPortImpl is initialized by the PortManagerImpl it reads its specific configuration data from the database, which includes the COM port name and the default modem init string. Because most configuration values exist in the base class and the base class provides methods that use these values, the base class setConfig method is called to store the configuration values in the base class. [*Note:* the normal way of doing this would be to call the base class constructor during construction, however because the InstallablePorts are instantiated generically by the PortManagerImpl, the constructors are not afforded the opportunity to take varying arguments.]

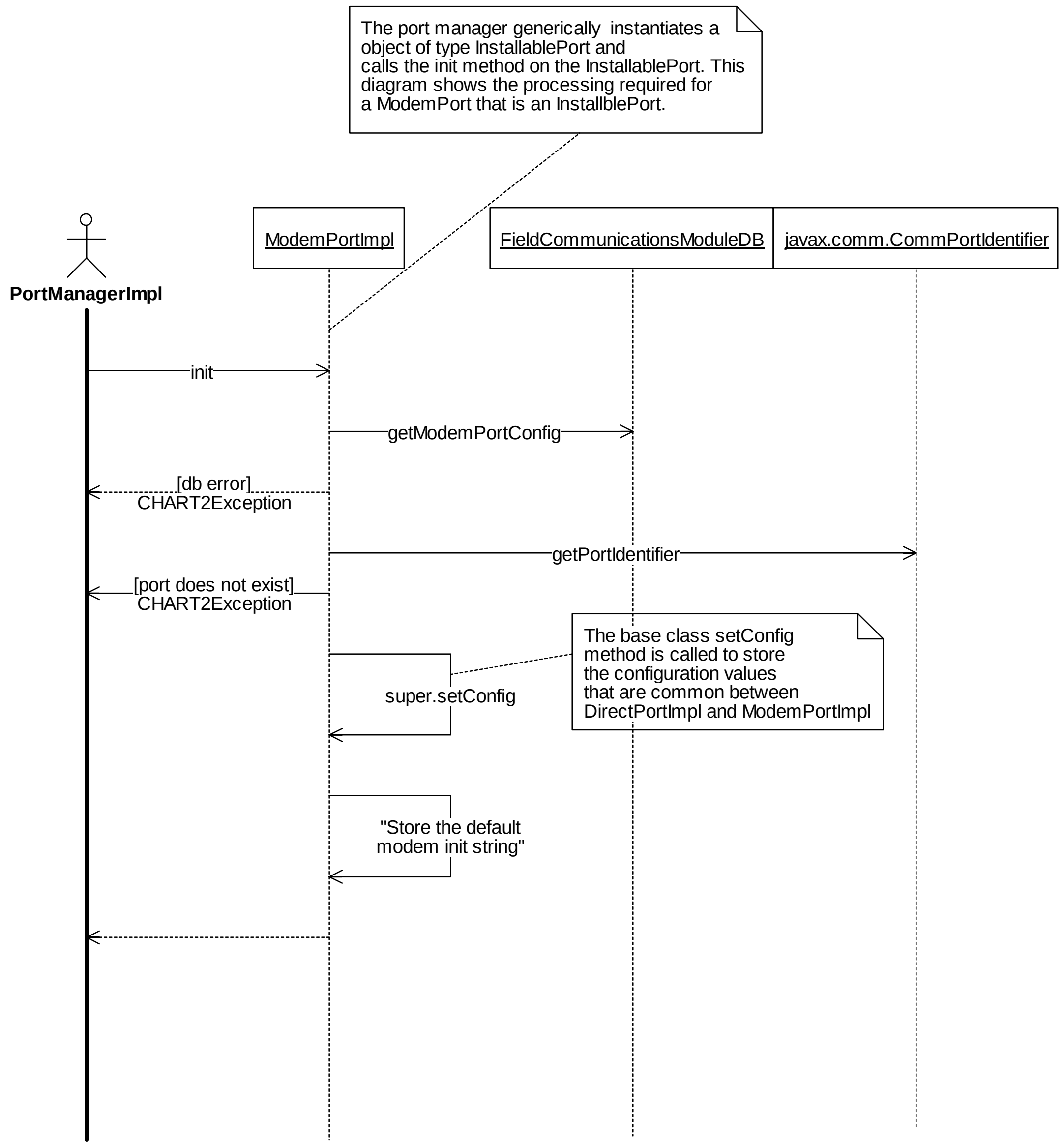

The port manager generically instantiates a object of type InstallablePort and calls the init method on the InstallablePort. This diagram shows the processing required for a ModemPort that is an InstallblePort.
PortManagerImpl
ModemPortImpl
FieldCommunicationsModuleDB
javax.comm.CommPortIdentifier
init
getModemPortConfig
[db error] CHART2Exception
getPortIdentifier
[port does not exist] CHART2Exception
super.setConfig
The base class setConfig method is called to store the configuration values that are common between DirectPortImpl and ModemPortImpl
"Store the default modem init string"

Figure 17. ModemPortImpl:init (Sequence Diagram)

3.2.2.14 ModemPortImpl:shutdown (Sequence Diagram)

When a ModemPortImpl is shutdown by the PortManagerImpl, it disconnects if it is currently connected.

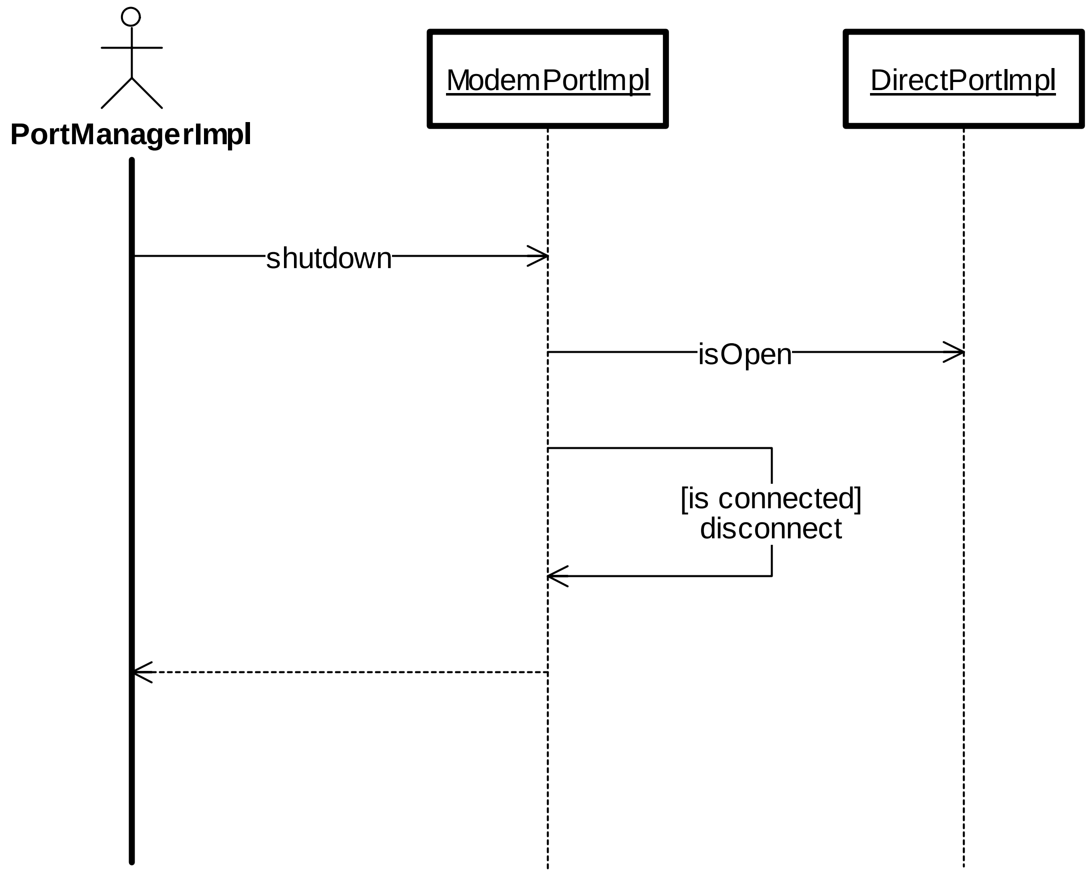
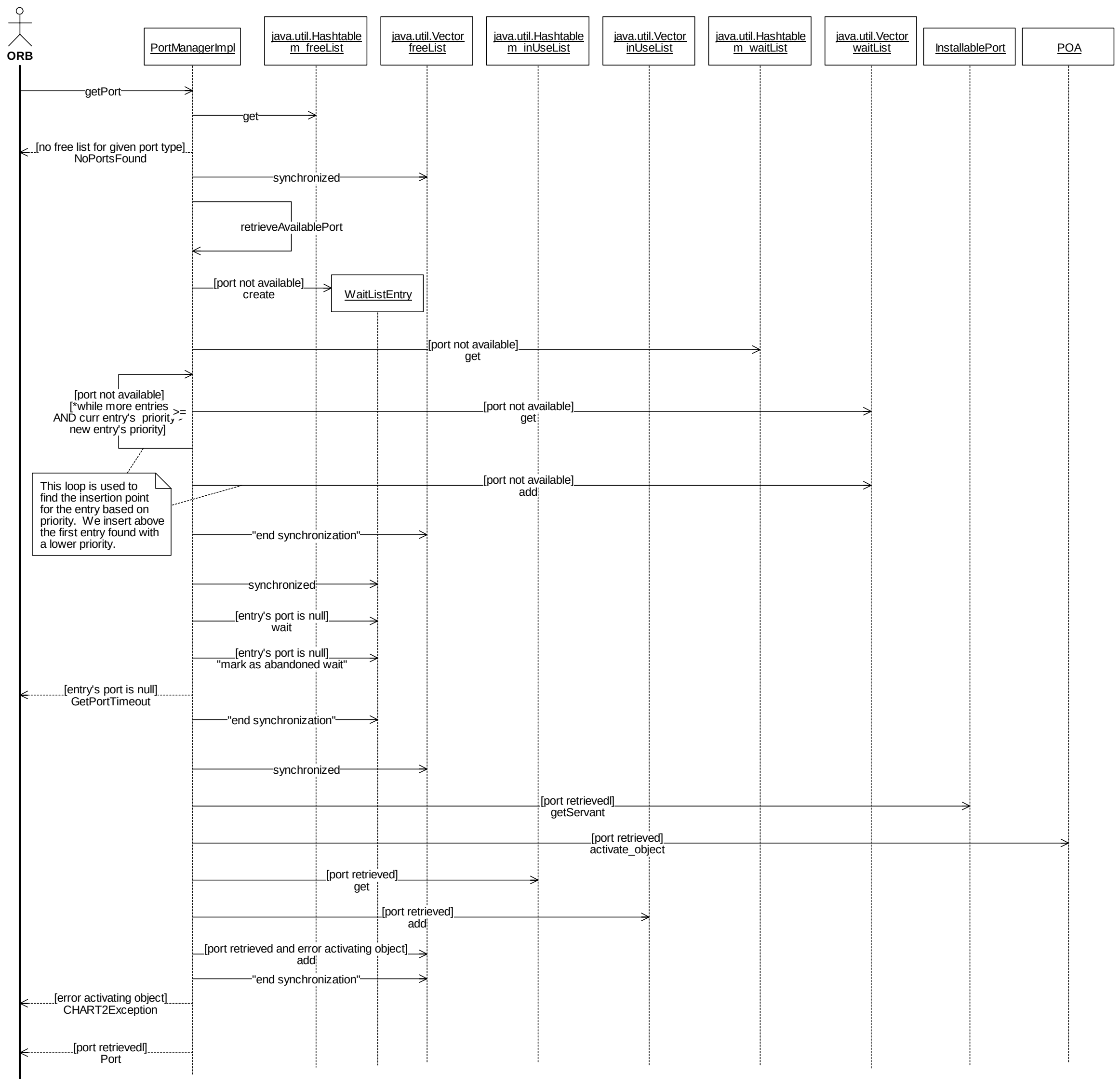


Figure 18. ModemPortImpl:shutdown (Sequence Diagram)

3.2.2.15 PortManagerImpl:getPort (Sequence Diagram)

When a request to get a port is received by the PortManagerImpl, it retrieves an available port (see PortManagerImpl:retrieveAvailablePort), activates it with the POA to make it available for CORBA calls, and returns the Port to the requester. In the event that a port is not available, getPort method creates a WaitList entry and inserts the entry into the wait list based on the priority of the request, using an insertion sort to keep the list ordered by order of decreasing priority and a secondary ordering of *fifo* based on the time added to the list. After adding an entry to the wait list, the getPort method waits on the entry's monitor for the releasePort method to notify it that a port has been handed off. If not notified within the timeout specified by the requester, the getPort method marks its wait list entry as abandoned and returns an exception to the requester.



ORB
PortManagerImpl
java.util.Hashtable m_freeList
java.util.Vector freeList
java.util.Hashtable m_inUseList
java.util.Vector inUseList
java.util.Hashtable m_waitList
java.util.Vector waitList
InstallablePort
POA
getPort
get
[no free list for given port type] NoPortsFound
synchronized
retrieveAvailablePort
[port not available] create
WaitListEntry
[port not available] get
[port not available] [*while more entries AND curr entry's priority >= new entry's priority]
[port not available] get
[port not available] add
This loop is used to find the insertion point for the entry based on priority. We insert above the first entry found with a lower priority.
"end synchronization"
synchronized
[entry's port is null] wait
[entry's port is null] "mark as abandoned wait"
[entry's port is null] GetPortTimeout
"end synchronization"
synchronized
[port retrievedl] getServant
[port retrieved] activate_object
[port retrieved] get
[port retrieved] add
[port retrieved and error activating object] add
"end synchronization"
[error activating object] CHART2Exception
[port retrievedl] Port

Figure 19. PortManagerImpl:getPort (Sequence Diagram)

3.2.2.16 PortManagerImpl:ReclaimPorts (Sequence Diagram)

The PortManagerImpl contains a Timer that periodically calls the PortManagerImpl's reclaimPorts method. The PortManagerImpl checks each port in its inUseList to see if it meets its own criteria for being deemed inactive. If a port is found to be inactive, it is deactivated from the POA, preventing any further calls to the port by its current user. The inactive port is then removed from the inUseList and returned to the freeList.

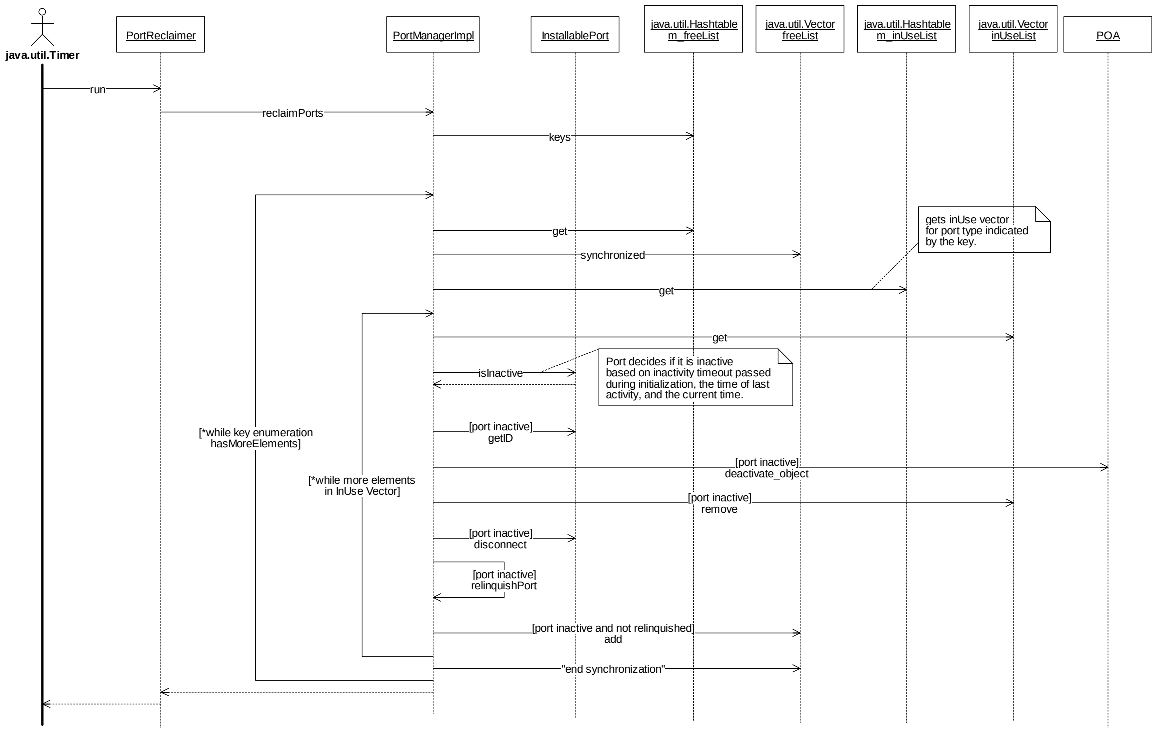

java.util.Timer
PortReclaimer
PortManagerImpl
InstallablePort
java.util.Hashtable m_freeList
java.util.Vector freeList
java.util.Hashtable m_inUseList
java.util.Vector inUseList
POA
run
reclaimPorts
keys
get
synchronized
get
gets inUse vector for port type indicated by the key.
get
isInactive
Port decides if it is inactive based on inactivity timeout passed during initialization, the time of last activity, and the current time.
[*while key enumeration hasMoreElements]
[port inactive] getID
[port inactive] deactivate_object
[*while more elements in InUse Vector]
[port inactive] remove
[port inactive] disconnect
[port inactive] relinquishPort
[port inactive and not relinquished] add
"end synchronization"

Figure 20. PortManagerImpl:ReclaimPorts (Sequence Diagram)

3.2.2.17 PortManagerImpl:ReleasePort (Sequence Diagram)

When a Port is released, the PortManagerImpl finds the port in its inUseList, disconnects the port and deactivates the object from the POA. The private relinquishPort method is called to hand off the port to the highest priority requester of the given port type. If there was no one waiting for the port, the port is removed from the inUseList and returned to the freeList. See the PortManagerImpl:reqlinquishPort sequence diagram for details on the hand off process.

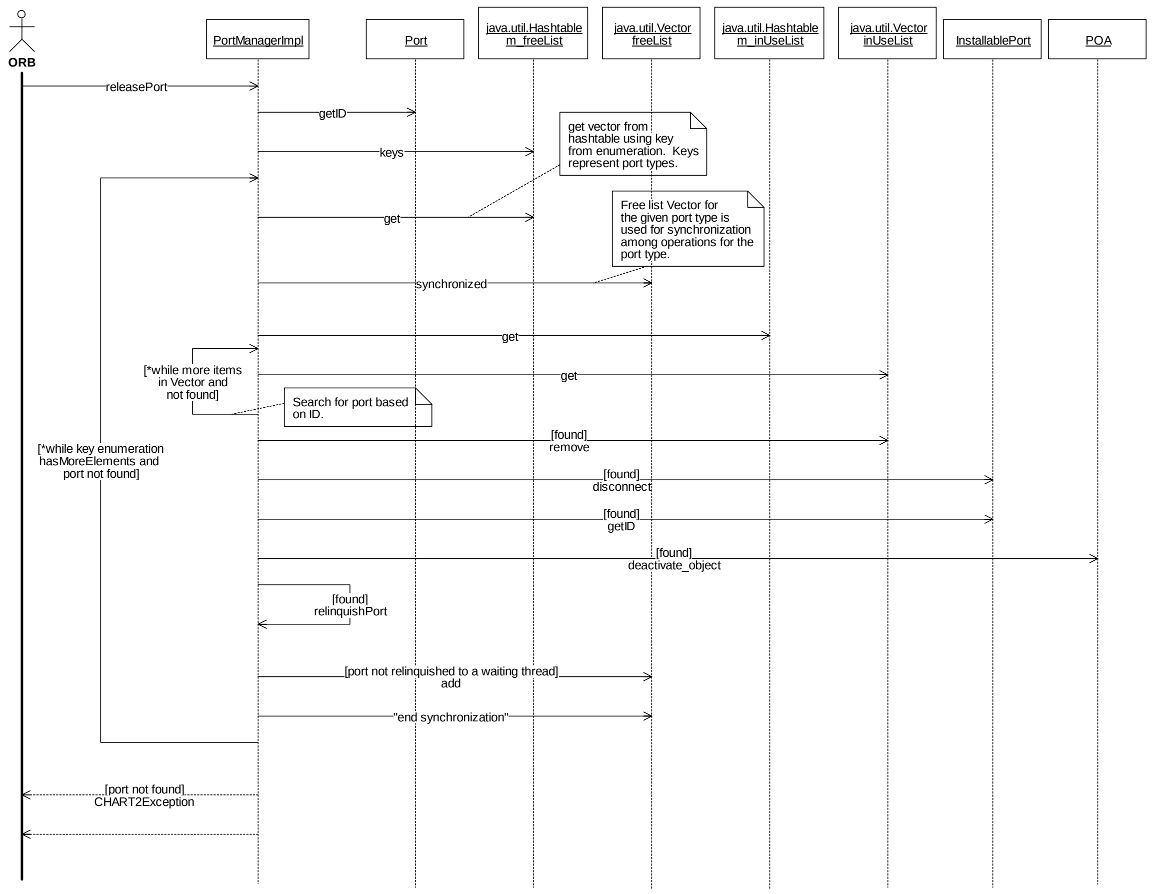

ORB
PortManagerImpl
Port
java.util.Hashtable m_freeList
java.util.Vector freeList
java.util.Hashtable m_inUseList
java.util.Vector inUseList
InstallablePort
POA
releasePort
getID
keys
get vector from hashtable using key from enumeration. Keys represent port types.
get
Free list Vector for the given port type is used for synchronization among operations for the port type.
synchronized
get
[*while more items in Vector and not found]
get
Search for port based on ID.
[*while key enumeration hasMoreElements and port not found]
[found] remove
[found] disconnect
[found] getID
[found] deactivate_object
[found] relinquishPort
[port not relinquished to a waiting thread] add
"end synchronization"
[port not found] CHART2Exception

Figure 21. PortManagerImpl:ReleasePort (Sequence Diagram)

3.2.2.18 PortManagerImpl:RelinquishPort (Sequence Diagram)

The PortManagerImpl relinquishPort method is a private helper method used to “hand off” a releasedPort to the top priority waiter (if any). The port is passed to the waiter through the WaitListEntry object that the waiter placed on the wait list. It is possible that a waiter put an entry on the wait list and then timed out. When this occurs the waiter marks the entry as abandoned. When the relinquishPort method encounters such entries, it simply removes them from the wait list and attempts to give the port to the next waiter in the list.

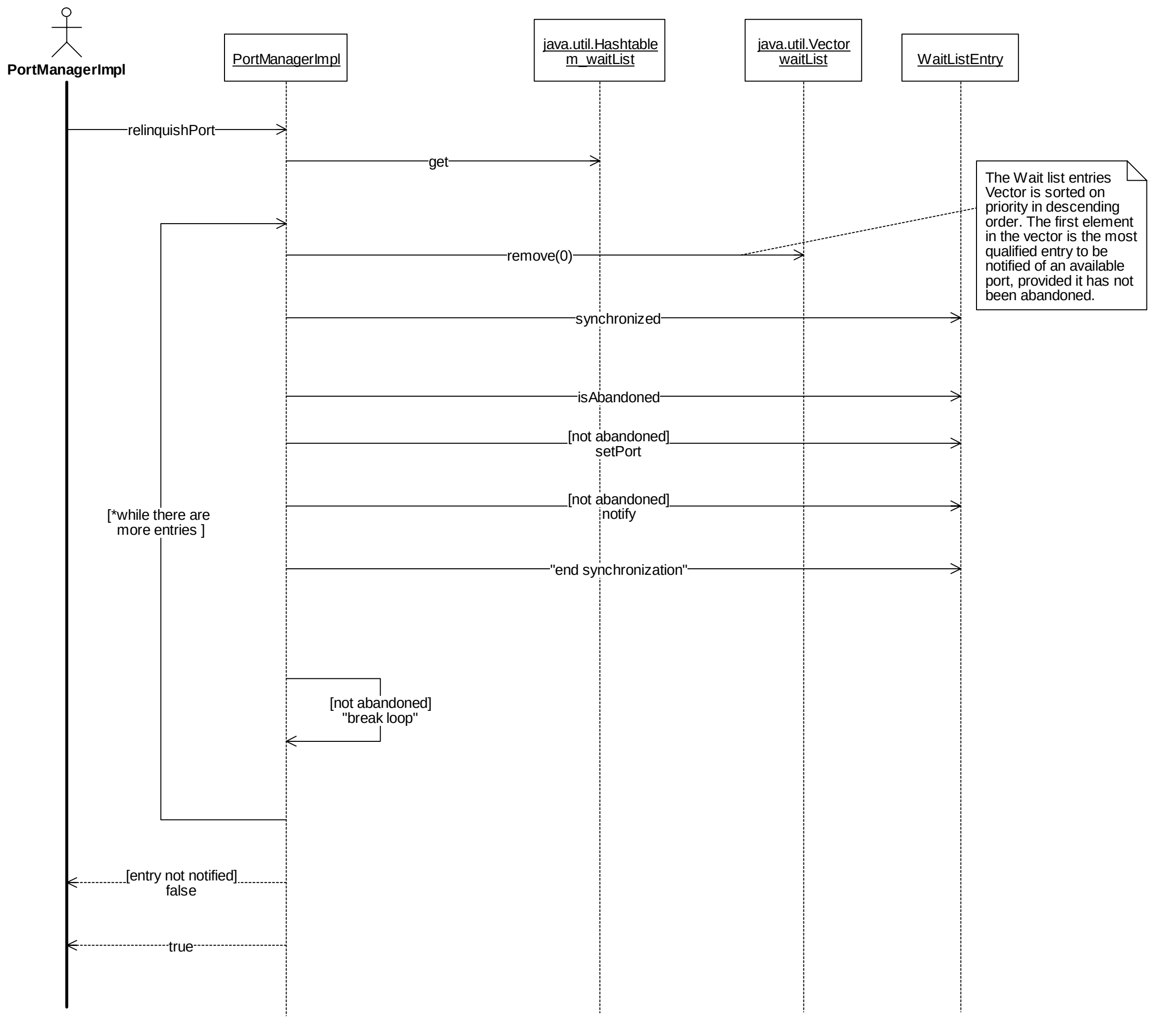
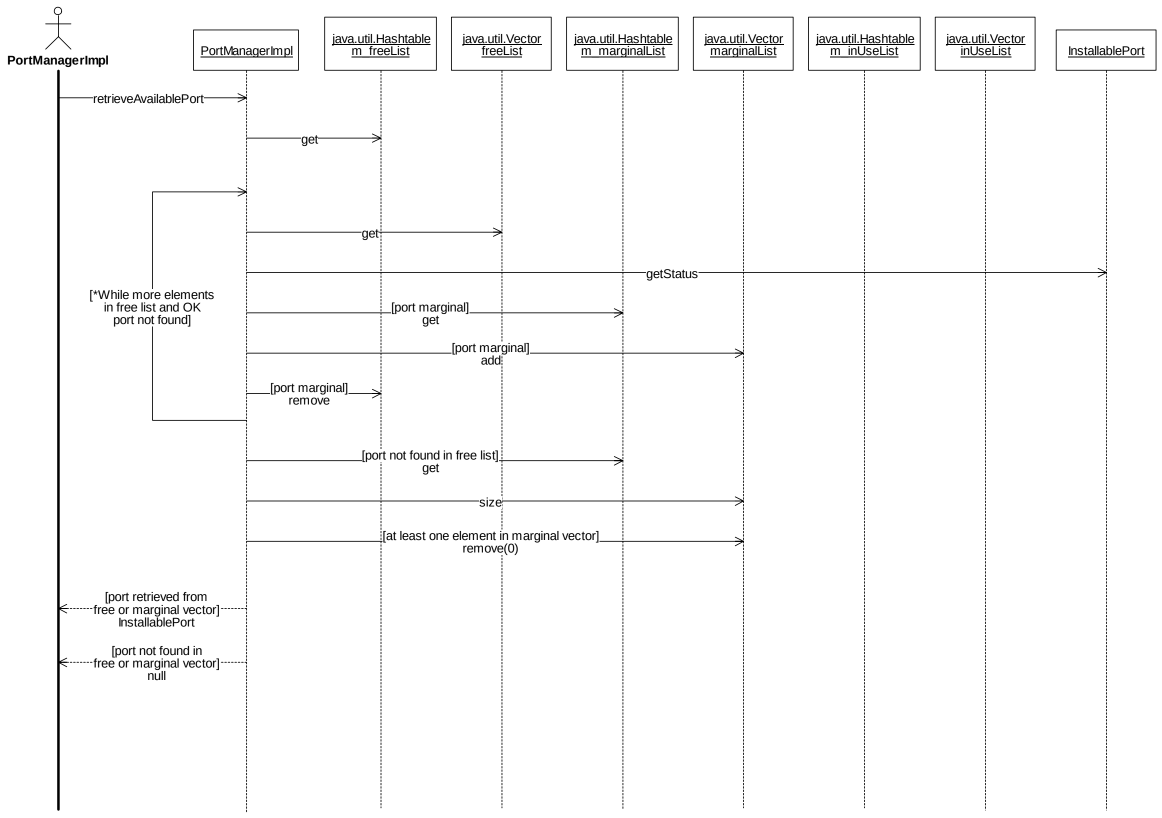

PortManagerImpl
PortManagerImpl
java.util.Hashtable m_waitList
java.util.Vector waitList
WaitListEntry
relinquishPort
get
The Wait list entries Vector is sorted on priority in descending order. The first element in the vector is the most qualified entry to be notified of an available port, provided it has not been abandoned.
remove(0)
synchronized
isAbandoned
[not abandoned] setPort
[*while there are more entries]
[not abandoned] notify
"end synchronization"
[not abandoned] "break loop"
[entry not notified] false
true

Figure 22. PortManagerImpl:RelinquishPort (Sequence Diagram)

3.2.2.19 PortManagerImpl:retrieveAvailablePort (Sequence Diagram)

The PortManagerImpl's retrieveAvailablePort method is a private helper method that manages removing a port from the free or marginal list, placing it in the inUseList, and returning the port. While searching for a port in the free list, any ports encountered that do not have a status of OK are moved to the end of the marginal list. Ports in the marginal list are only retrieved if a port is not available in the free list.



PortManagerImpl
PortManagerImpl
java.util.Hashtable m_freeList
java.util.Vector freeList
java.util.Hashtable m_marginalList
java.util.Vector marginalList
java.util.Hashtable m_inUseList
java.util.Vector inUseList
InstallablePort
retrieveAvailablePort
get
get
getStatus
[*While more elements in free list and OK port not found]
[port marginal] get
[port marginal] add
[port marginal] remove
[port not found in free list] get
size
[at least one element in marginal vector] remove(0)
[port retrieved from free or marginal vector] InstallablePort
[port not found in free or marginal vector] null

Figure 23. PortManagerImpl:retrieveAvailablePort (Sequence Diagram)

3.3 DMS Protocols

3.3.1 Classes

3.3.1.1 Protocol Handler Classes

This class diagram shows the protocol handler classes that are related to DMS control.

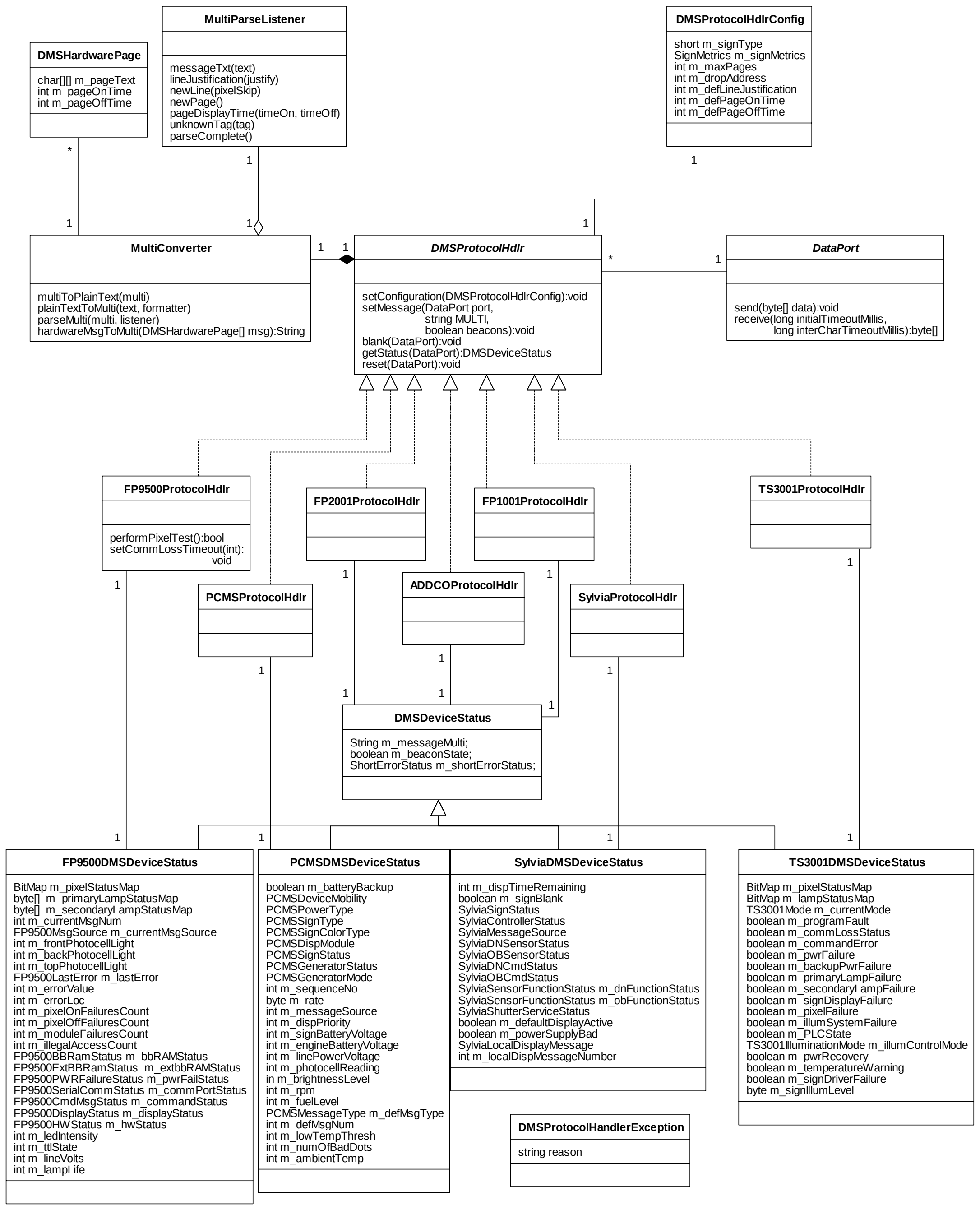


Figure 24. DMSProtocols (Class Diagram)

3.3.1.1.1 ADDCOProtocolHdlr (Class)

This protocol handler contains the protocol for communicating with an ADDCO portable DMS.

3.3.1.1.2 DataPort (Class)

A DataPort is a port that allows binary data to be sent and received. Ports of this type support a receive method that allows a chunk of all available data to be received. This method prevents callers from having to issue many receive calls to parse a device response. Instead, this receive call returns all available data received within the timeout parameters. The caller can then parse the data within a local buffer. Using this mechanism, device command and response should require only one call to send and one call to receive.

3.3.1.1.3 DMSDeviceStatus (Class)

This class contains data returned by all DMS protocol handlers getStatus() method. DMSs that support more detailed status return a derivation of this class.

3.3.1.1.4 DMSHardwarePage (Class)

This class holds data that specifies the layout of one page of a DMS message on the actual DMS hardware. A two dimensional array that is the same size as the sign's display (rows and columns) specifies the character displayed in each cell, including blank if the cell has no character. This format maps well to the way DMS protocols return the current message being displayed in a status query. This class can then be passed to a MultiConverter object to convert the message into MULTI format.

3.3.1.1.5 DMSProtocolHandlerException (Class)

This exception is thrown when a DMS device fails to respond to a command or a protocol error is detected in the response packet.

3.3.1.1.6 DMSProtocolHdlr (Class)

This interface defines the methods that must be supported by DMS protototocol handlers. [*Note:* some handlers support methods in addition to these standard methods.]

3.3.1.1.7 FP9500ProtocolHdlr (Class)

This protocol handler implements the protocol used to command an FP9500 DMS. The performPixelTest method causes a pixel test to be run on the sign. The status of pixels reported in the getStatus method contains the status since the last time a pixel test was run.

3.3.1.1.8 MultiConverter (Class)

This class provides methods which perform conversions between the DMS MULTI mark-up language and plain text. It also provides a method that will parse a MULTI message and inform a MultiParseListener of elements found in the message.

3.3.1.1.9 PCMSDMSDeviceStatus (Class)

This class contains status data that is returned from the Display Solutions PCMS protocol handler in the getStatus call.

3.3.1.1.10 DMSProtocolHdlrConfig (Class)

This class contains the configuration parameters for the DMS Protocol handlers.

3.3.1.1.11 SylviaDMSDeviceStatus (Class)

This class contains status data that is returned from the Sylvia protocol handler in the getStatus call.

3.3.1.1.12 FP1001ProtocolHdlr (Class)

This protocol handler contains the protocol used to communicate with an FP1001 DMS.

3.3.1.1.13 FP2001ProtocolHdlr (Class)

This protocol handler contains the protocol used to communicate with an FP2001 DMS.

3.3.1.1.14 PCMSProtocolHdlr (Class)

This protocol handler contains the protocol used to communicate with a Display Solutions (Winkomatic) Portable DMS.

3.3.1.1.15 SylviaProtocolHdlr (Class)

This protocol handler contains the protocol used to communicate with a Sylvia DMS.

3.3.1.1.16 MultiParseListener (Class)

A MultiParseListener works in conjunction with the MultiConverter to allow an implementing class to be notified as parsing of a MULTI message occurs. An exemplary use of a MultiParseListener would be the MessageView window which will need to have the MULTI message parsed in order to display it as a pixmap.

3.3.1.1.17 FP9500DMSDeviceStatus (Class)

This class contains status data that is returned from the FP9500 protocol handler in the getStatus call.

3.3.1.1.18 TS3001DMSDeviceStatus (Class)

This class contains data returned from the TS3001 protocol handler's getStatus() method.

3.3.1.1.19 TS3001ProtocolHdlr (Class)

This protocol handler contains the protocol used to communicate with a Telespot 3001 series DMS.

3.3.1.2 Support Classes

This diagram contains the support classes used by the various DMS protocol handlers to provide extended status reporting.

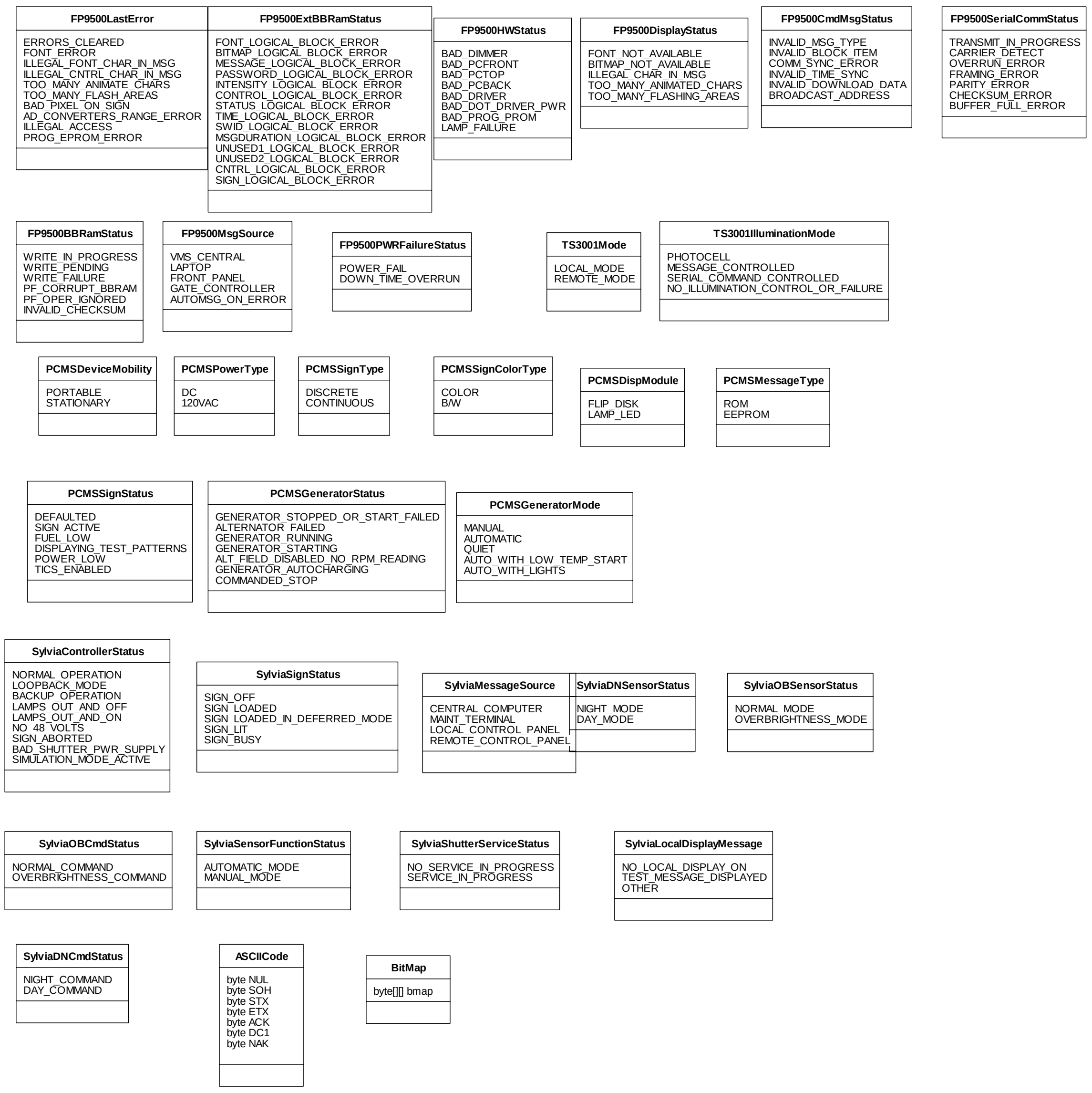


Figure 25. ProtocolSupportClasses (Class Diagram)

3.3.1.2.1 ASCIICode (Class)

This class is a holder for ASCII codes used by protocol handlers when communicating with a DMS.

3.3.1.2.2 BitMap (Class)

This structure is used to pass status data that maps to pixels on a DMS, such as pixel status or lamp status. Each row of the bmap member corresponds to a row of pixels on the DMS. A value of 1 in a cell indicates the status for that pixel is OK while a zero indicates a failure.

3.3.1.2.3 FP9500BBRamStatus (Class)

This enumeration defines the valid values for the Battery Backed RAM Status in a FP9500 device.

3.3.1.2.4 FP9500PWRFailureStatus (Class)

This enumeration defines the valid values that indicate the power failure condition in a FP9500 device.

3.3.1.2.5 FP9500DisplayStatus (Class)

This enumeration defines the valid values that indicate the message error status of a previous message display operation on a FP9500 device.

3.3.1.2.6 FP9500HWStatus (Class)

This enumeration defines the valid values that indicate the sign controller hardware error status of a FP9500 device.

3.3.1.2.7 FP9500LastError (Class)

This enumeration defines the reasons for the failure of the last device command sent to a FP9500 device.

3.3.1.2.8 FP9500MsgSource (Class)

This enumeration defines the valid values for a originator of the current message displayed on a FP9500 device.

3.3.1.2.9 FP9500CmdMsgStatus (Class)

This enumeration defines the valid values that indicate the status of the message selection command sent to a FP9500 device.

3.3.1.2.10 FP9500ExtBBRamStatus (Class)

This enumeration defines the values that indicate a corrupt logical block that was reported as a result of Battery backed RAM error on a FP9500 device.

3.3.1.2.11 FP9500SerialCommStatus (Class)

This enumeration defines the valid values that indicate the serial communication port status of the FP9500 device.

3.3.1.2.12 PCMSDeviceMobility (Class)

This enumeration defines the valid values that indicate the mobility type of a Display solutions PCMS device.

3.3.1.2.13 PCMSDispModule (Class)

This enumeration defines the valid values that indicate the type of display module used in a Display Solutions PCMS device.

3.3.1.2.14 PCMSGeneratorMode (Class)

This enumeration defines the valid values that indicate the Generator mode of the Display Solutions PCMS device.

3.3.1.2.15 PCMSGeneratorStatus (Class)

This enumeration defines the valid values that indicate the Generator status of a Display Solutions PCMS device.

3.3.1.2.16 PCMSMessageType (Class)

This enumeration defines the valid values that indicate the various message types used in a Display Solutions PCMS device.

3.3.1.2.17 PCMSPowerType (Class)

This enumeration defines the valid values that indicate the Power type of a Display Solutions PCMS device.

3.3.1.2.18 PCMSSignColorType (Class)

This enumeration defines the valid values that indicate the color of a Display Solutions PCMS device.

3.3.1.2.19 PCMSSignStatus (Class)

This enumeration defines the valid values that indicate the Sign status of a Display Solutions PCMS device.

3.3.1.2.20 PCMSSignType (Class)

This enumeration defines the valid values that indicate the sign module type of a Display Solutions PCMS device.

3.3.1.2.21 SylviaDNSensorStatus (Class)

This enumeration defines the valid values for the Day/Night Sensor status of a Sylvia device.

3.3.1.2.22 SylviaOBSensorStatus (Class)

This enumeration defines the valid values for the Overbrightness Sensor status of a Sylvia device.

3.3.1.2.23 SylviaDNCmdStatus (Class)

This enumeration defines the valid values for the Day/Night command status of a Sylvia device.

3.3.1.2.24 TS3001Mode (Class)

This enumeration defines the operational modes of a TS3001 device.

3.3.1.2.25 SylviaControllerStatus (Class)

This enumeration defines the valid values that indicate the controller status of a Sylvia device.

3.3.1.2.26 SylviaMessageSource (Class)

This enumeration defines the valid values for a originator of the current message displayed on a Sylvia device.

3.3.1.2.27 TS3001IlluminationMode (Class)

This enumeration defines the valid values that indicate the sign illumination control setting on a TS3001 device.

3.3.1.2.28 SylviaOBCmdStatus (Class)

This enumeration defines the valid values for the Overbrightness command status of a Sylvia device.

3.3.1.2.29 SylviaSignStatus (Class)

This enumeration defines the valid values that indicate the sign module status of a Sylvia device.

3.3.1.2.30 SylviaLocalDisplayMessage (Class)

This enumeration defines the valid values for the local display message of a Sylvia device.

3.3.1.2.31 SylviaSensorFunctionStatus (Class)

This enumeration defines the valid values for the sensor function status of a Sylvia device.

3.3.1.2.32 SylviaShutterServiceStatus (Class)

This enumeration defines the valid values for the shutter service status of a Sylvia device.

3.3.2 Sequence Diagrams

3.3.2.1 DMSProtocols:TypicalSetMessage (Sequence Diagram)

This sequence shows typical processing of a protocol handler to set the message of a DMS. All protocol handlers have slightly different implementations due to the different protocols being implemented. However, all protocol handlers have a general goal of formatting a byte array according to the device protocol, sending the byte array to the device, and receiving a response from the device to determine if the command was successful. Because DMS messages are specified in the MULTI format, part of the processing required to format a byte array to command the DMS includes converting the MULTI message into the proper sequence of bytes the DMS expects. The MultiConverter class helps to parse through the MULTI tags and pull apart the message into simple pieces that the protocol handler can use to format the byte array. Once told to parse a multi string, the MultiConverter calls back into the parse listener (which happens to be the protocol handler in our case) as it encounters multi tags and message text. After the protocol handler has formatted the byte array, it sends it to the device using the DataPort interface, which may actually be a modem or a direct connect port. After sending the command, the protocol handler reads the response from the device and determines if the command was successful. Failures are indicated though the use of exceptions which contain a specific reason for the failure.

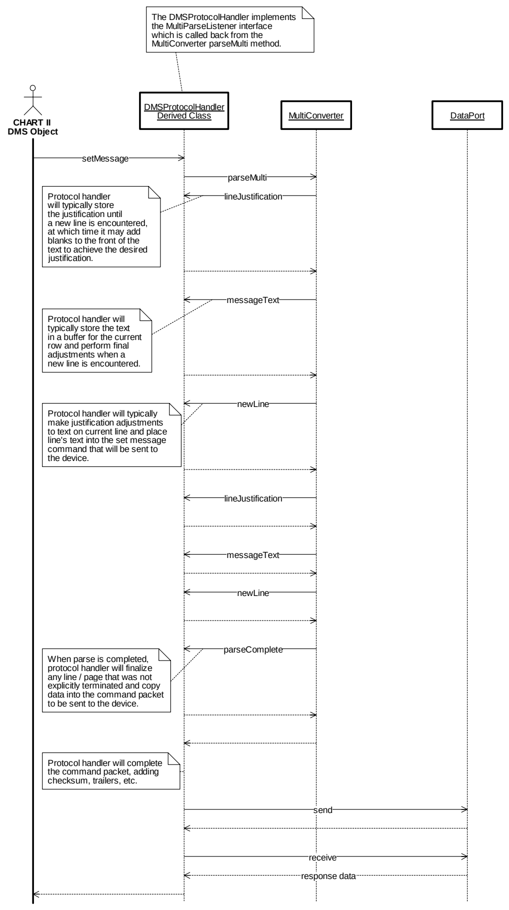
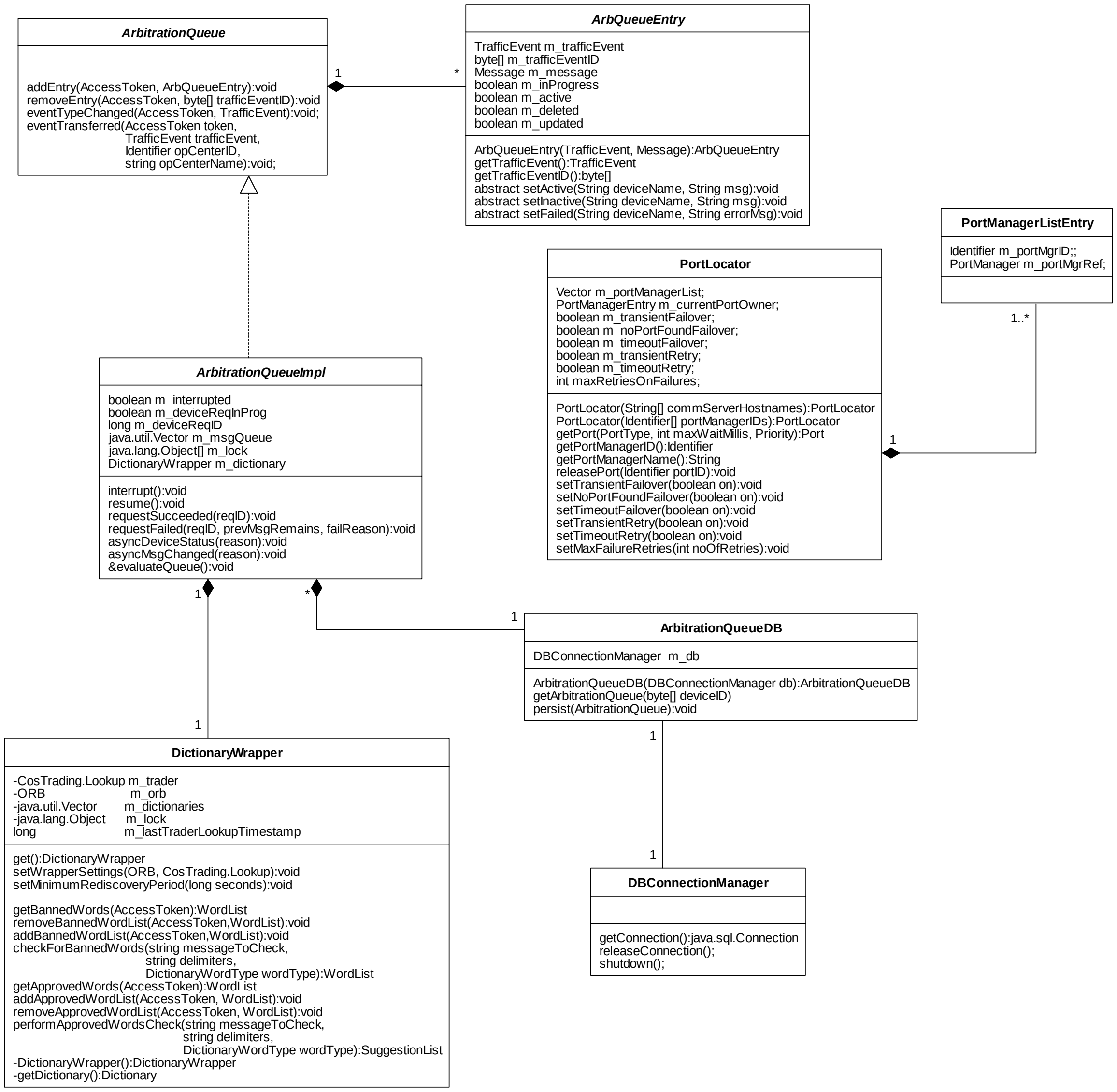

The DMSProtocolHandler implements the MultiParseListener interface which is called back from the MultiConverter parseMulti method.
CHART II DMS Object
DMSProtocolHandler Derived Class
MultiConverter
DataPort
setMessage
parseMulti
lineJustification
Protocol handler will typically store the justification until a new line is encountered, at which time it may add blanks to the front of the text to achieve the desired justification.
messageText
Protocol handler will typically store the text in a buffer for the current row and perform final adjustments when a new line is encountered.
newLine
Protocol handler will typically make justification adjustments to text on current line and place line's text into the set message command that will be sent to the device.
lineJustification
messageText
newLine
parseComplete
When parse is completed, protocol handler will finalize any line / page that was not explicitly terminated and copy data into the command packet to be sent to the device.
Protocol handler will complete the command packet, adding checksum, trailers, etc.
send
receive
response data

Figure 26. DMSProtocols:TypicalSetMessage (Sequence Diagram)

3.4 Device Utility

3.4.1 Classes

This diagram shows the classes contained in the DeviceUtility package. These classes are used mainly by CHART II device objects. The classes PortLocator and PortManagerListEntry have been added to simplify use of failover with the FMS subsystem.



ArbitrationQueue
addEntry(AccessToken, ArbQueueEntry):void
removeEntry(AccessToken, byte[] trafficEventID):void
eventTypeChanged(AccessToken, TrafficEvent):void;
eventTransferred(AccessToken token, TrafficEvent trafficEvent, Identifier opCenterID, string opCenterName):void;
1
*
ArbQueueEntry
TrafficEvent m_trafficEvent
byte[] m_trafficEventID
Message m_message
boolean m_inProgress
boolean m_active
boolean m_deleted
boolean m_updated
ArbQueueEntry(TrafficEvent, Message):ArbQueueEntry
getTrafficEvent():TrafficEvent
getTrafficEventID():byte[]
abstract setActive(String deviceName, String msg):void
abstract setInactive(String deviceName, String msg):void
abstract setFailed(String deviceName, String errorMsg):void
PortManagerListEntry
Identifier m_portMgrID;;
PortManager m_portMgrRef;
1..*
PortLocator
Vector m_portManagerList;
PortManagerEntry m_currentPortOwner;
boolean m_transientFailover;
boolean m_noPortFoundFailover;
boolean m_timeoutFailover;
boolean m_transientRetry;
boolean m_timeoutRetry;
int maxRetriesOnFailures;
PortLocator(String[] commServerHostnames):PortLocator
PortLocator(Identifier[] portManagerIDs):PortLocator
getPort(PortType, int maxWaitMillis, Priority):Port
getPortManagerID():Identifier
getPortManagerName():String
releasePort(Identifier portID):void
setTransientFailover(boolean on):void
setNoPortFoundFailover(boolean on):void
setTimeoutFailover(boolean on):void
setTransientRetry(boolean on):void
setTimeoutRetry(boolean on):void
setMaxFailureRetries(int noOfRetries):void
1
ArbitrationQueueImpl
boolean m_interrupted
boolean m_deviceReqInProg
long m_deviceReqID
java.util.Vector m_msgQueue
java.lang.Object[] m_lock
DictionaryWrapper m_dictionary
interrupt():void
resume():void
requestSucceeded(reqID):void
requestFailed(reqID, prevMsgRemains, failReason):void
asyncDeviceStatus(reason):void
asyncMsgChanged(reason):void
&evaluateQueue():void
1
*
1
ArbitrationQueueDB
DBConnectionManager m_db
ArbitrationQueueDB(DBConnectionManager db):ArbitrationQueueDB
getArbitrationQueue(byte[] deviceID)
persist(ArbitrationQueue):void
1
1
1
DictionaryWrapper
-CosTrading.Lookup m_trader
-ORB m_orb
-java.util.Vector m_dictionaries
-java.lang.Object m_lock
long m_lastTraderLookupTimestamp
get():DictionaryWrapper
setWrapperSettings(ORB, CosTrading.Lookup):void
setMinimumRediscoveryPeriod(long seconds):void
getBannedWords(AccessToken):WordList
removeBannedWordList(AccessToken,WordList):void
addBannedWordList(AccessToken,WordList):void
checkForBannedWords(string messageToCheck, string delimiters, DictionaryWordType wordType):WordList
getApprovedWords(AccessToken):WordList
addApprovedWordList(AccessToken, WordList):void
removeApprovedWordList(AccessToken, WordList):void
performApprovedWordsCheck(string messageToCheck, string delimiters, DictionaryWordType wordType):SuggestionList
-DictionaryWrapper():DictionaryWrapper
-getDictionary():Dictionary
DBConnectionManager
getConnection():java.sql.Connection
releaseConnection();
shutdown();

Figure 27. DeviceUtility (Class Diagram)

3.4.1.1.1 PortLocator (Class)

The PortLocator is a utility class that helps one to utilize the fault tolerance provided by the deployment of many PortManagers. The PortLocator is initialized by specifying a preferred PortManager and optionally one or more alternate PortManagers. When asked to acquire a port, the PortLocator first attempts to acquire a port from the preferred PortManager and falls back to alternate PortManager objects when faults occur. The PortLocator can also be set to determine the fallback action (if any) if a Port cannot be obtained from the preferred PortManager.

The failure types that may occur are:

1. The PortManager does not have any of the requested type of ports (either none exist or all that it has are failed).
2. The PortManager’s ports of the requested type are all in use and one does not become available within the timeout specified.
3. A connection to the PortManager object cannot be established (CORBA.Transient, CORBA.ObjectNotExist). When one of these failures occurs, the PortLocator uses its settings to determine if it will attempt to acquire a port from the next communication server in its list or return an error to the caller.

Because the PortLocator is initialized with port manager object IDs or PortManager names, it uses the CORBA trader to obtain object references for the PortManager on each communication server.

The PortLocator is designed to be used by a single device object. Only one port may be requested at a time, thus a second call to getPort prior to a call to releasePort will result in an exception. When the PortLocator acquires a port for the user it stores the PortManager from which it received the port and can provide the name or ID (depending on how the PortLocator was initialized) of the PortManager from which the port was retrieved.

3.4.1.1.2 PortManagerListEntry (Class)

This class is used by the PortLocator to map object identifiers to object references for PortManager objects.

3.4.2 Sequence Diagrams

3.4.2.1 PortLocator:getPort (Sequence Diagram)

When a request is made to the PortLocator to get a port, the PortLocator gets the first entry from its port manager list (which is the preferred port manager) and asks the port manager for a port. If the getPort call on the port manager fails, the PortLocator consults its retry settings and may retry the operation depending on the specific failure condition. If all retries (if any) of the getPort operation on the port manager are exhausted without success, the PortLocator may failover to the next PortManager in the PortLocator’s list, depending on the specific error condition encountered and the settings for failover in the PortLocator. Because the PortLocator is initialized with only object identifiers for the preferred and fallback port managers, a Trader query is made to obtain an object reference the first time a PortManager is to be accessed.

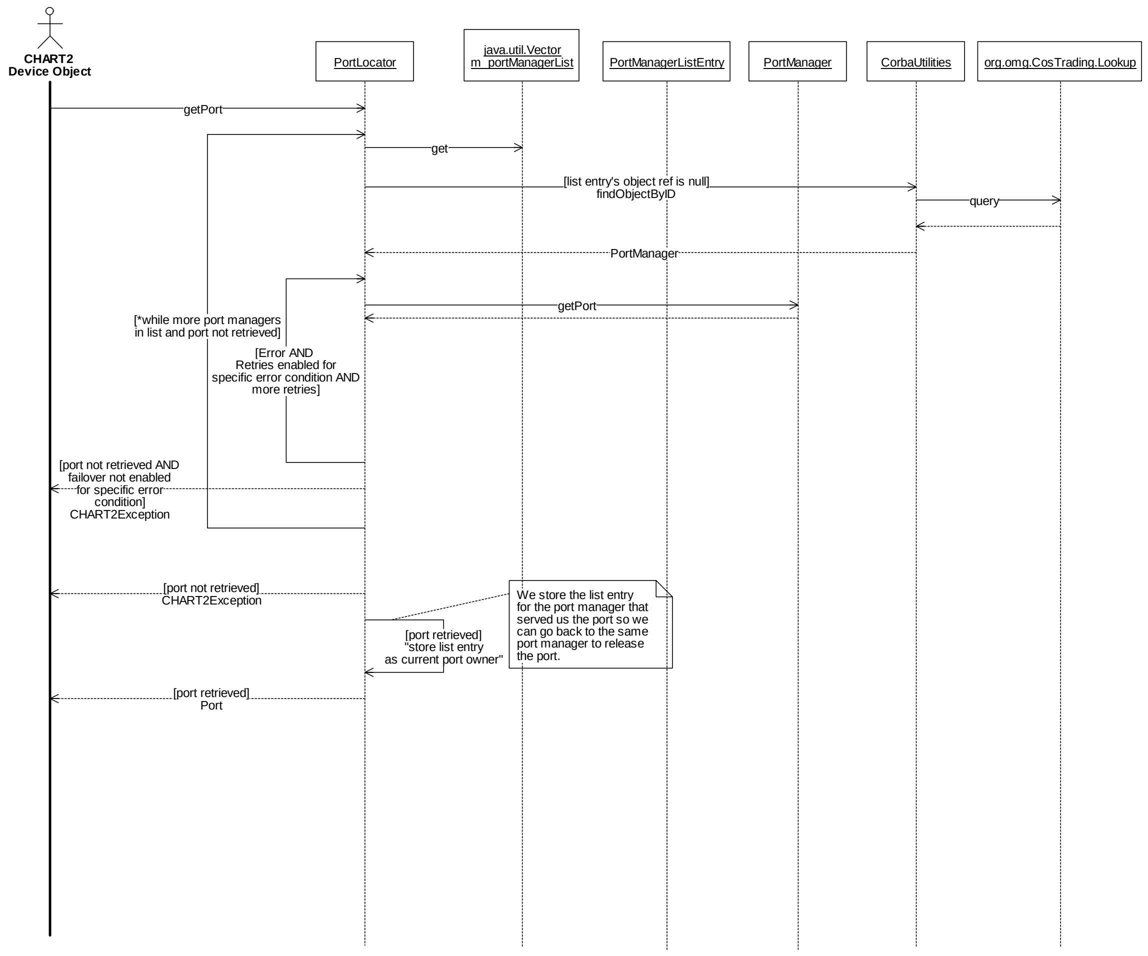


CHART2 Device Object
PortLocator
java.util.Vector m_portManagerList
PortManagerListEntry
PortManager
CorbaUtilities
org.omg.CosTrading.Lookup
getPort
get
[list entry's object ref is null] findObjectByID
query
PortManager
getPort
[*while more port managers in list and port not retrieved]
[Error AND Retries enabled for specific error condition AND more retries]
[port not retrieved AND failover not enabled for specific error condition] CHART2Exception
[port not retrieved] CHART2Exception
[port retrieved] "store list entry as current port owner"
We store the list entry for the port manager that served us the port so we can go back to the same port manager to release the port.
[port retrieved] Port

Figure 28. PortLocator:getPort (Sequence Diagram)

3.4.2.2 PortLocator:ReleasePort (Sequence Diagram)

When the PortLocator releasePort method is called, the PortLocator uses the port manager reference that it stored in the getPort method to release the port from the correct PortManager.

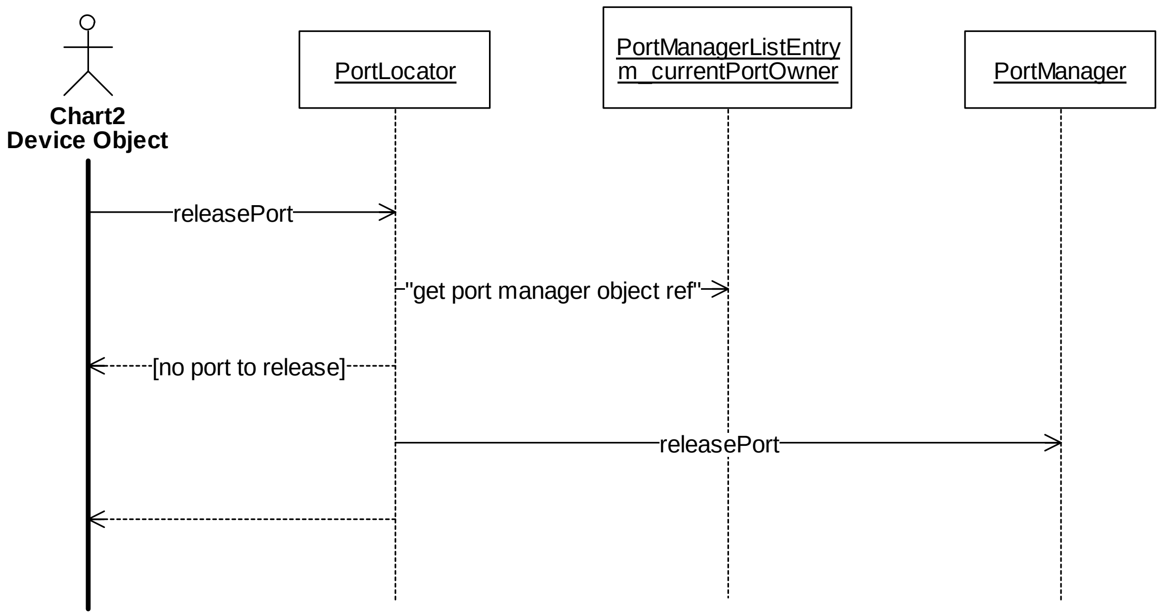


Figure 29. PortLocator:ReleasePort (Sequence Diagram)

Bibliography

CHART II Business Area Architecture Report, document number M361-BA-005R0, Computer Sciences Corporation and PB Farradyne, Inc., April 28, 2000

CHART II System Requirements Specification Release 1 Build 2, document no. M361-RS-002R1, Computer Sciences Corporation and PB Farradyne, Inc.

FMS R1B2 High Level Design, document number M303-DS-002R0, Computer Sciences Corporation and PB Farradyne, Inc.

The Common Object Request Broker: Architecture and Specification, Revision 2.3.1, OMG Document 99-10-07

Martin Fowler and Kendall Scott, *UML Distilled*, Addison-Wesley, 1997

TELE-SPOT 3001 Sign Controller Communications Protocol, document no. 750208-040 v2.3, T-S Display Systems Inc., 1995

Functional Specification for FP9500ND – MDDOT Display Control System, document no. A316111-080 Rev. A6, MARK IV Industries Ltd., 1998.

Maintenance Manual for the FP1001 Display Controller, document no. 316000-443 Rev. E, Ferranti-Packard Displays, 1987

FP2001 Display Controller Application Guide, document no. A317875-012 Rev. 8, F-P Electronics, 1991

Engineering Specification - Brick Sign Communications Protocol, Rev. 1, ADDCO Inc., 1999.

PCMS Protocol version 4, document number 32000-150 Rev. 5, Display Solutions, 2000

BSC Protocol Specification (Data Link Protocol Layer), v. 1.3, Fiberoptic Display Systems Inc., 1996

Sylvia Variable Message Sign, Command Set 9403-1, v. 1.4, Fiberoptic Display Systems Inc., 1996

2.5 Mile AM Travelers Information Station Instruction Manual For: Maryland State Highway Administration, Information Station Specialists.

Technical Practice RC-2A Remote Touch-Tone On/Off Industrial Controller, Viking Electronics Inc., August 1993.

Acronymns

The following acronyms appear throughout this document:

API	Application Program Interface
BAA	Business Area Architecture
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DMS	Dynamic Message Sign
FMS	Field Management Station
GUI	Graphical User Interface
HAR	Highway Advisory Radio
IDL	Interface Definition Language
ISDN	Integrated Services Digital Network
ITS	Intelligent Transportation Systems
LATA	Local Access and Transport Areas
MULTI	Mark Up Language for Transportation Information
NTCIP	National Transportation Communications for ITS Protocol
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
POTS	Plain Old Telephone System
R1B2	Release 1, Build 2 of the CHART II System
TTS	Text To Speech
UML	Unified Modeling Language

Appendix A – Glossary

COM Port (or COMM Port)	A serial communications port on a computer. These ports are typically named COM1, COM2, etc.
CORBA	An object oriented software architecture that allows software objects to interact over a network.
CORBA Event	A CORBA mechanism using which different Chart2 components exchange information without explicitly knowing about each other.
CORBA Trader	A CORBA service that facilitates object location and discovery. A server advertises an object in the Trading Service based on the kind of service provided by the object. A client locates objects of interest by asking the Trading Service to find all objects that provide a particular service.
Direct Port	A type of Port that provides access directly to a COM port on the machine that serves the DirectPort object.
DMS	A Dynamic Message Sign that can be controlled by one Operations Center at a time.
Factory	A CORBA object that is capable of creating other CORBA objects of a particular type. The newly created object will be served from the same process as the factory object that creates it.
FMS	Field Management Station through which the CHART II system communicates with the devices in the field.
Graphical User Interface	Part of a software application that provides a graphical interface to its user.
HAR	A radio transmitter used to broadcast traffic information to the public.
Installable Module	A software object that can be included in an application through the configuration of the application.
ModemPort	A type of Port that provides access to a modem.

MULTI	An NTCIP standard mark up language used to specify the display of DMS messages.
MultiConverter	A software utility object used to convert text to and from the MULTI format.
Object Discovery	A GUI mechanism in which the client periodically asks the CORBA Trading Service to find objects of those types that are of interest to the GUI, such as DMS, HAR, Plan etc.
Operations Center	A center where one or more users may log in to operate the CHART II system. Operations centers are assigned responsibility for shared resources that are controlled by users who are logged in at that operations center.
Operator	A CHART II user that works at an Operations Center.
Port	A CORBA object used to generically represent a single communications resource available on a computer. Derived interfaces define functionality specific to the type of communications resource.
PortManager	A CORBA object used by clients to gain access to Port objects. The PortManager manages access to pre-configured Port objects and allows ports to be shared amongst many clients.
ProtocolHandler	A software object that contains code that is knowledgeable of the protocol used to command a specific make and model of a device.
Service Application	A software application that can be configured to run one or more service application modules and provides them basic services needed to serve CORBA objects.
Service Application Module	A software module that serves a related group of CORBA objects and can be run within the context of a service application.
System Version	ObjectTeam/Cool:Jex term used to describe a logical grouping of related design work products. Examples of system versions created in support of this design include FieldCommunicationsModule and DMSProtocols.

SHAZAM	The name given to a type of roadside sign which advises motorists to tune to a radio station to hear traffic information when beacons on the sign are flashing.
Sign	see DMS
User	A user is anyone who uses the CHART II system. A user can perform different operations in the system depending upon the roles they have been granted.